НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Факультет інформатики та обчислювальної техніки
(повне найменування інституту, факультету)

кафедра обчислювальної техніки
(повна назва кафедри)

До захисту допущено
Завідувач кафедри

_____ *С.Г. Стіренко*
(підпис) (ініціали, прізвище)

“___” _____ 2020 р.

Дипломний проект

на здобуття ступеня бакалавра

з напряму підготовки 6.050123 «Комп’ютерна інженерія»

на тему *«Мікросервісна система моніторингу фільмів»*

Виконав: студент *4* курсу, групи *ІО-63*
Здота Віктор Олександрович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник *асист. Калюжний О. О.* _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Консультант основна частина *асист. Калюжний О. О.* _____
(назва розділу) (вчені ступінь та звання, прізвище, ініціали) (підпис)

Консультант нормоконтроль *проф. Сімоненко В. П.* _____
(назва розділу) (вчені ступінь та звання, прізвище, ініціали) (підпис)

Рецензент _____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проекті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ - 2020 року

Національний технічний університет України
«Київський політехнічний інститут»

Інститут (факультет) інформатики та обчислювальної техніки
(повна назва)
Кафедра обчислювальної техніки
(повна назва)

Рівень вищої освіти – перший (бакалаврський)

Спеціальність 6.050123 – « Комп'ютерна інженерія»
(повна назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри

С.Г. Стіренко
"___" _____ 2020 року

ЗАВДАННЯ
на дипломний проект студента

Здоти Віктора Олександровича
(прізвище, ім'я, по батькові)

1. Тема проекту *«Мікросервісна система моніторингу фільмів»*

керівник проекту *Калюжний О. О.*,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "___" _____ 20__ року N ___

2. Термін подання студентом проекту _____

3. Вихідні дані до проекту технічна документація, теоретичні дані, інтернет-публікації за темою роботи

4. Зміст пояснювальної записки

- *Провести огляд мікросервісної архітектури;*
- *Провести аналіз інструментів необхідних для розробки системи;*
- *Спроектувати та розробити мікросервісну систему моніторингу фільмів;*
- *Провести тестування розробленої системи;*

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо)

6. Консультанти розділів проекту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	*асист. Калюжний О. О.*	*01.11.2019*	*01.11.2019*
2	*асист. Калюжний О. О.*	*01.11.2019*	*01.11.2019*
3	*асист. Калюжний О. О.*	*01.11.2019*	*01.11.2019*
4	*асист. Калюжний О. О.*	*01.11.2019*	*01.11.2019*

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту	Строк виконання етапів проекту	Примітка
1	*Затвердження теми роботи*	*01.09.2019*	*виконано*
2	*Вивчення та аналіз завдання*	*15.10.2019-15.11.2019*	*виконано*
3	*Проведення огляд мікросервісної архітектури*	*16.11.2019-20.12.2019*	*виконано*
4	*Проведення аналізу інструментів необхідних для розробки системи*	*21.12.2029-19.01.2020*	*виконано*
5	*Проектування та розробка мікросервісної системи моніторингу фільмів*	*20.01.2020-31.03.2020*	*виконано*
6	*Проведення тестування розробленої системи*	*01.04.2020-25.04.2020*	*виконано*
7	*Оформлення матеріалів роботи*	*25.04.2020-25.05.2020*	*виконано*
8	*Передзахист*	*26.05.2020*	
9	*Захист*	...	

Студент _____ Здота В.О.
(підпис) (прізвище та ініціали)

Керівник проекту (роботи) _____ *Калюжний О. О.*
(підпис) (прізвище та ініціали)

№ рядка	Формат	Позначення	Найменування	Кількість	Примітка
1	А4		Завдання на дипломний проект	2	
2	А4	ІАЛЦ.006310.001 ВП	Відомість дипломного проекту	1	
3	А4	ІАЛЦ.006310.002 ТЗ	Технічне завдання	3	
4	А4	ІАЛЦ.006310.003 ПЗ	Пояснювальна записка	68	
5	А3	ІАЛЦ.006310.004 Д1	Алгоритм оновлення фільмів. Схема принципова.	1	
6	А3	ІАЛЦ.006310.005 Д2	Схема зв'язків у базах даних. Схема функціонльна.	1	
7	А3	ІАЛЦ.006310.006 Д3	Взаємодія частин системи. Схема структурна.	1	

Зм.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.006310.001 ВП		
Розробив		Здота В.О.			Мікросервісна система моніторингу фільмів	Літ.	Аркуш 1 / Аркушів 1
Перевір.		Калюжний О.О.					
Н. контр.		Сімоненко В.П.			Відомість дипломного проекту	НТУУ "КПІ", ФІОТ, ІО-63	
Затверд.							

Технічне завдання до дипломного проекту

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання розповсюджується на розробку і користування мікросервісної системи моніторингу фільмів.

Область застосування: Створення веб застосунку для перегляду фільмів з функціоналом для додання нових фільмів зі стороннього ресурсу.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки служить покращення масштабованості веб застосунків, які призначені для роботи з фільмами.

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проекту є розробка мікросервісної системи моніторингу фільмів.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелами для розробки служать науково-технічна література з комп'ютерних технологій, публікації в періодичних виданнях, документація до бібліотек та технологій, публікації в Інтернеті за даним питанням.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розроблюваного продукту

- Розробка декількох мікросервісів для забезпечення процесу оновлення фільмів.
- Забезпечення незалежності мікросервісів.
- Розробка центрального API системи.
- Розробка клієнтського застосунку для візуалізації роботи з системою.

5.2. Вимоги до програмного забезпечення

- Операційна система MS Windows XP, MS Windows Vista, MS Windows 7, MS Windows 8/8.1, MS Windows 10, Linux, MacOS
- Docker 3.5 і вище
- Microsoft SQL Server 2014 і вище

5.3. Вимоги до апаратного забезпечення

- Комп’ютер на базі процесору Intel Core 5 і вище
- Оперативної пам’яті не менше 8 Гбайт

Пояснювальна записка до дипломного проекту

на тему: Мікросервісна система моніторингу фільмів

Київ - 2020 року

ЗМІСТ

ВСТУП

На сьогодні прогрес змушує програми виконувати велику кількість різноманітних задач. Тому сучасні системи та застосунки є багатофункціональними, з доволі складною логікою та архітектурою. Великі монолітні програми, тобто ті, що є єдиним неподільним цілим, не завжди приносять бажаний результат. Зосередження логіки в одному місці може призвести до складнощів при розробці, надмірних витрат ресурсів і грошей та до складнощів у підтримці продукту.

В даній роботі буде розроблена система моніторингу фільмів. Процес моніторингу буде здійснено за допомогою періодичного опитування стороннього ресурсу і розсилки відповідних повідомлень. Так як система моніторингу фільмів буде складною в плані різноплановості і повинна бути розширюваною за потреби, є необхідність в пошуку доцільної архітектури, відмінної від моноліту.

Одним із рішень даної проблеми є виокремлення частин логіки і розподілення обов'язків між меншими частинами системи, що в ідеальному варіанті призведе до утворення мікросервісної архітектури. Як було сказано вище при такому підході бізнес логіка ділиться між окремими частинами, які називають мікросервісами, кожна з яких має відповідати за рішення певної задачі. В такому випадку на етапі розробки різними мікросервісами можуть займатися різні команди програмістів, не знаючи про деталі реалізації інших.

Головним завданням перед архітекторами таких систем є забезпечення передачі інформації між мікросервісами. Одним із способів реалізації цього є використання проміжних вершин, що будуть служити для доставки повідомлень між частинами системи. Такі вершини називаються брокерами. Прикладом найпопулярніших слугують ActiveMQ та Kafka. Найбільшою перевагою у їх використанні є відв'язка сервісів один від одного. Єдина сутність з якою спілкується мікросервіс є брокер, тож йому нічого не відомо про інші вершини в системі.

РОЗДІЛ 1
ОГЛЯД МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

1.1. Мікросервісна архітектура та основні принципи

Мікросервісна архітектура (MSA – Microservice Architecture) - це структура призначена для розробки програмного забезпечення, що включає слабко пов'язані модульні компоненти, відомі як мікросервіси[1].

Мікросервіс - це дискретний компонент більшого сервісу, який підтримує конкретну бізнес-мету і використовує простий, чітко визначений інтерфейс для спілкування з іншими мікросервісами[14]. Модульність мікросервісів дозволяє масштабувати окремі компоненти програми окремо відповідно до вимог та мінімізує ймовірність того, що зміна, внесена в один програмний елемент, створить непередбачені зміни в інших елементах. Мікросервіси можна додавати, видаляти, перейменовувати, переналаштовувати, змінювати та переставляти, не впливаючи ні на одного, ні на програму в цілому[2].

Чітких обов’язкових принципів, яким має слідувати мікросервіс не має, так як це залежить від конкретних потреб, але можна виділити ті властивості, що зустрічаються найчастіше[3].

Виділимо наступні властивості:

1) невеликий розмір;
2) незалежність;
3) будується навколо бізнес потреби та має обмежений контекст;
4) взаємодія з іншими мікросервісами виконується за рахунок шаблону Smart endpoints and dumb pipes (Розумні кінцеві точки, тобто мікросервіси, та дурні засоби передачі інформації, тобто брокери);
5) через розподілену структуру використання підходу Design for Failure є необхідністю;
6) зведення централізації зверху до мінімуму;

7) необхідність автоматизації в процесах розробки та підтримки;
8) Ітераційний розвиток.

1.1.1. Невеликий розмір мікросервіса

Невеликий розмір – не дуже інформативне формулювання. Це тому що кожен архітектор має сам визначитися з розміром. Адже під конкретні бізнес кейси не можливо зробити чітких стандартів, бо всі вони концептуально відрізняються.

Та існують деякі рекомендації експертів[1], що можуть бути використані для оцінки. Мікросервіс має бути такого розміру, щоб виконувалося одне з тверджень:

- один мікросервіс має розвиватися командою кількістю не більше, ніж з дюжину;
- команда кількістю півдюжини чоловік може такою ж кількістю мікросервісів;
- план розробки і бізнес логіки мікросервіса може поміститися в голову однієї людини;
- за одну Agile ітерацію одна команда може переписати один мікросервіс.

1.1.2. Незалежність мікросервісу

Мікросервісна архітектура має бути розроблена за принципами Low Coupling та High Cohesion (слабка зв’язність та висока згуртованість)[1]. По суті, висока згуртованість означає винесення частин коду, які пов'язані один з одним, в одне місце. У той же час низьке з'єднання полягає в тому, щоб якомога більше відокремити незалежні частини один від одного. Тобто кожен мікросервіс має містити в собі всю необхідну базу для розв’язку тієї чи іншої задачі, і в той самий час бути відокремленим від інших мікросервісів[15].

Введемо нове поняття.

Компонент - це одиниця програмного забезпечення, код якої може бути незалежно замінений або оновлений.

В сучасній розробці ПЗ будь-яка велика програма складається з компонентів, що відповідають за окремі частини бізнес логіки. Але так як монолітна архітектура представлена загальною кодовою базою, дотримуватися принципів Low Coupling та High Cohesion дуже складно і рано чи пізно вони будуть порушені.

В той же час розбивання системи на окремі сервіси зобов'язує дотримуватися жорстокого відокремлення.

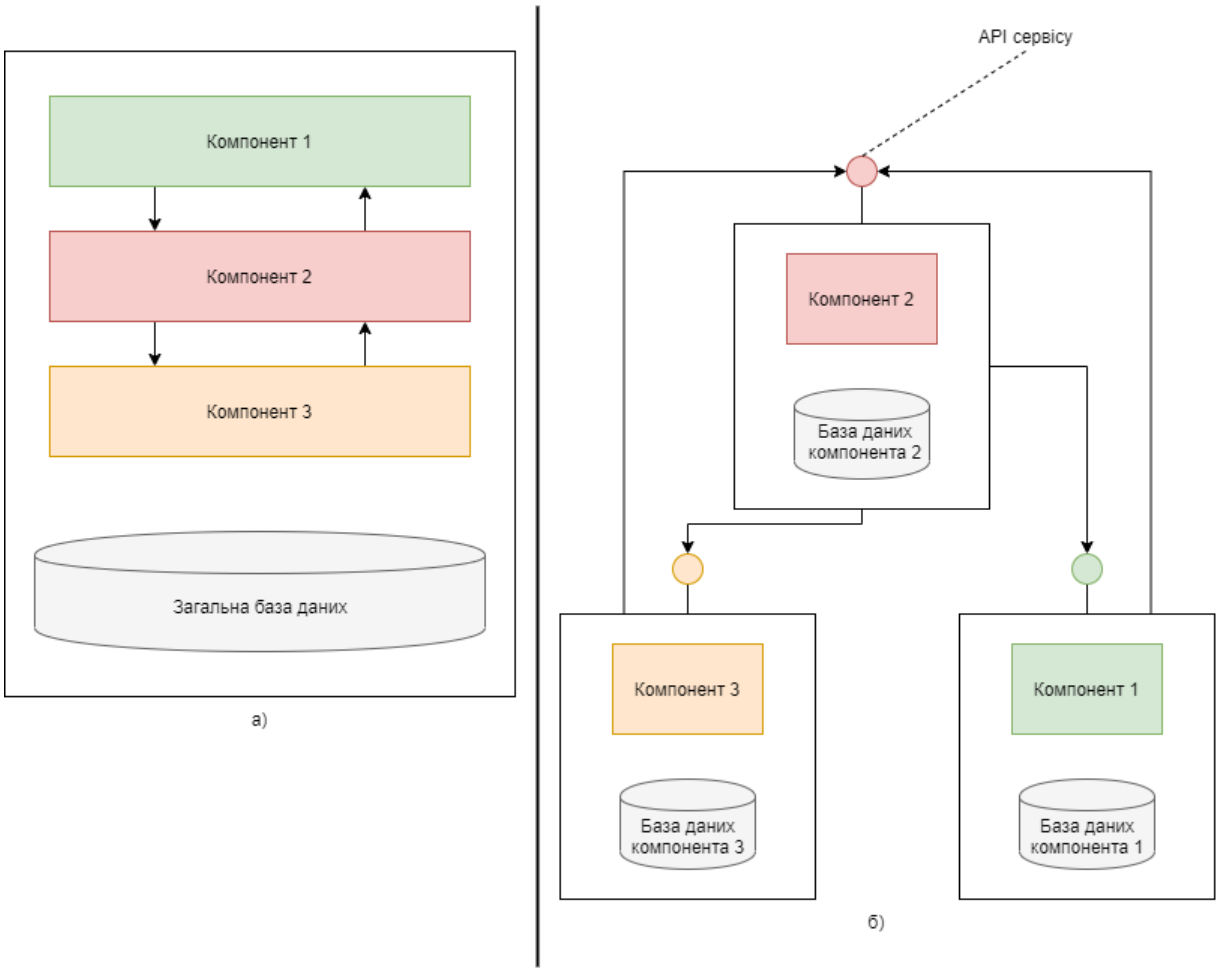


Рис. 1.1. а) – приклад монолітної архітектури; б) – приклад мікросервісної архітектури

На рисунку 1.1 зображено приклад монолітної (а) та мікросервісної архітектури (б). У першому випадку компоненти знаходяться в одному місці

і між ними існує явний зв'язок, у другому ж ці самі компоненти ізольовані один від одного і взаємодіють за допомогою інтерфейсів.

З вище сказаного можна зробити висновок, що кожен мікросервіс зобов’язаний бути незалежним компонентом. Так як кожен мікросервіс працює в своєму процесі, він має чітко винести інтерфейс взаємодії з ним (API). Інші вершини можуть взаємодіяти з сервісом лише через API, тому мінімізація зв’язків – один а найважливіших процесів в плануванні такої архітектури.

1.1.3. Виокремлення бізнес потреби у мікросервіс

При проектуванні нового мікросервісу найголовніший етап – це визначення його меж. Від цього залежатиме подальша розробка і це буде мати серйозний вплив на подальше розгортання та життєвий цикл сервісу.

Основний підхід до визначення цих самих меж – це виділити конкретну бізнес-потребу і сформувати зону відповідальності навколо неї. Чим компактніше бізнес потреба, чим більш формалізовані є її взаємовідносини з іншими частинами системи, тим простіше розробка сервісу. Найголовніше та найскладніше - це витримати зону відповідальності в подальшому процесі розробки та розвитку, тому слід якнайкраще продумати можливі майбутні зміни в бізнес логіці та врахувати їх при плануванні розробки[15].

Після виокремлення меж мікросервісу і винесення всіх його елементів в єдину кодову базу слід захистити його від зовнішнього впливу, такого як комутація з іншими вершинами системи. Далі створюється власне контекст мікросервісу, який може для будь-якого об’єкта чи будь-якої взаємодії мати власну, відмінну від інших частин програми інтерпретацію. Приклад різної інтерпретації контексту об’єкту Користувача в різних сервісах зображено на рисунку 1.2.

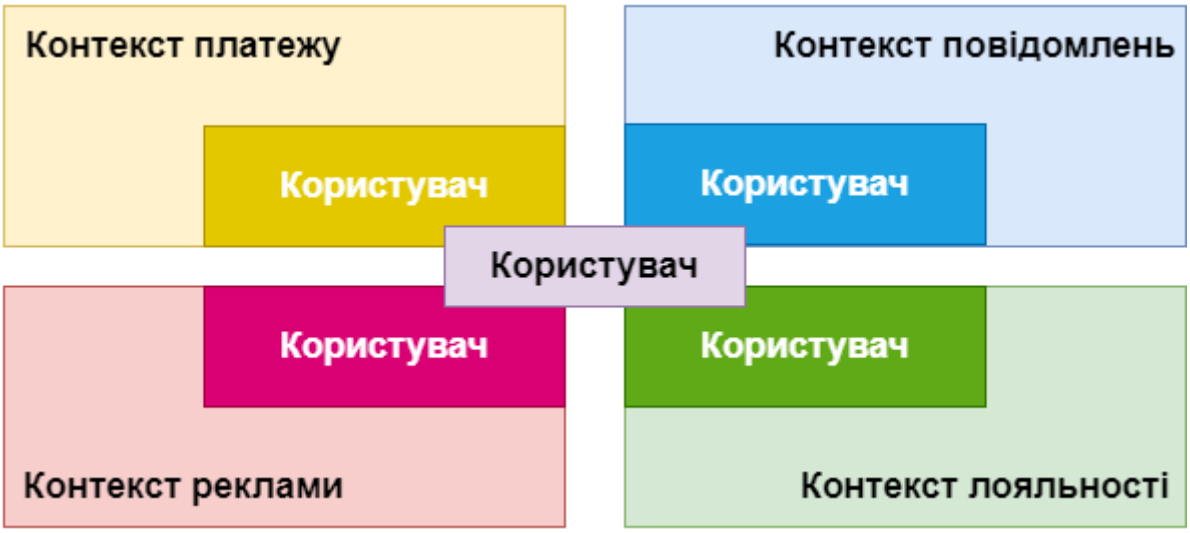


Рис 1.2 Різна інтерпретація контексту для одного об’єкту – користувача

Якщо межі було визначено не правильно, то як наслідок доведеться змінювати функціональність не лише даного мікросервісу а й інших. В результаті цього інтерфейси взаємодії з сервісами будуть змінені, що в свою чергу призведе майже до повного перероблювання системи, що у великих проектах спричинить серйозні витрати ресурсів та людино-годин.

Для зменшення можливості таких помилок треба заздалегідь продумати межі. Для цього існує підхід Monolith First (Спочатку Моноліт)[3]. Спочатку систему розробляють у вигляді моноліту, а коли з’являються відокремлювані області, то їх розносять по мікросервісах.

В результаті того, що мікросервіси відокремлені один від одного не лише на логічному, а й на технологічному рівні, можливо для різних потреб створювати мікросервіси в рамках однієї системи використовуючи різні мови програмування, бібліотеки та використовувати різні операційні системи.

1.1.4. Інтеграція шаблону Smart endpoints and dumb pipes

Інтеграція мікросервісів здійснюється без використання ESB (Enterprise Service Bus), що виступає центральною ланкою в багатьох сервісно-орієнтованих системах, так як ESB порушує принципи незалежності та

децентралізованості. Отже складність інтеграції розповсюджується з централізованої ланки (ESB) безпосередньо на компоненти (Smart endpoints - розумні кінцеві точки).

В більшості випадках для інтеграції використовують прості текстові протоколи, які є надбудовою над HTTP. Саме це дає змогу обійти технологічні відмінності окремих мікросервісів.

Також використовуються бінарні протоколи. Вони звісно є ефективнішими. Але вимагають від компонентів бути зв’язаними за кодовою базою, що суперечить принципу незалежності.

Незалежно від вибору протоколу передачі інформації вся логіка відправки та обробки повідомлень відбувається безпосередньо у мікросервісах.

1.1.5. Принцип Design for Failure

Розподілена природа мікросервісів, безперечно, приносить дуже багато переваг. В той же час такий підхід є критичним місцем. Адже вершини системи взаємодіють через мережу, яка за своєю сутністю не є надійною. Наприклад, вона може відмовити, не пропускати повідомлення через файрвол та інше.

Мікросервіси ж, які спілкуються через мережу, можуть почати працювати повільніше або взагалі перестати відповідати. Тож при кожному віддаленому виклику необхідно це враховувати. З рисунку 3.1, на якому зображено приклад розподіленої системи з декількома зв’язками, видно, що при відмові одного з компонентів системи, всі ті, яким необхідна його робота можуть також вийти з нормального робочого режиму, якщо це не було враховано при проектуванні. Якщо є необхідність, то мікросервіс має зачекати на відповідь або належним чином опрацювати відмову та повернутися до роботи.

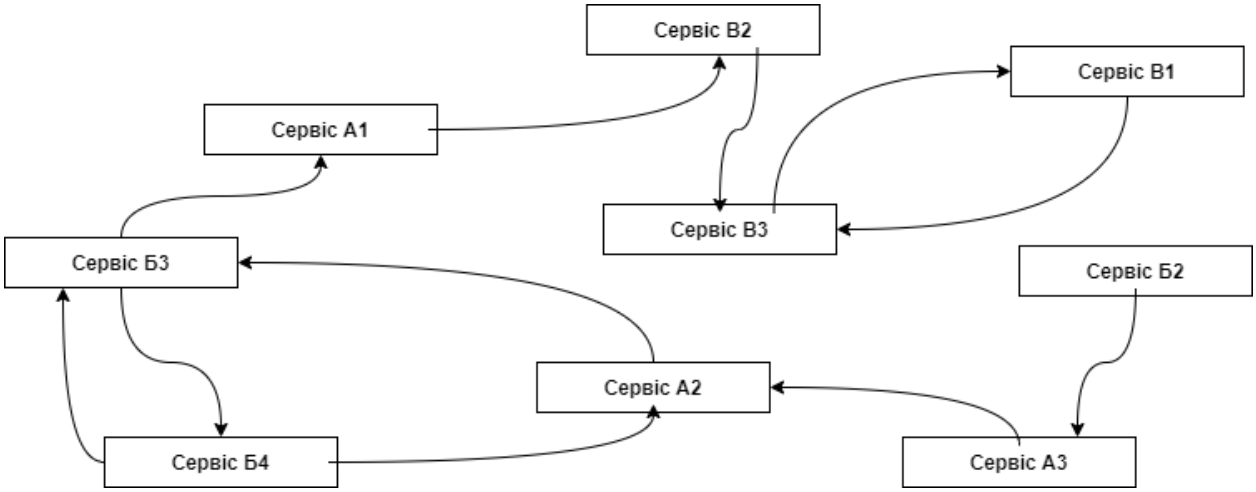


Рис 1.3 Приклад системи зав’язків між мікросервісами

1.1.6. Децентралізація даних

Децентралізація даних ще один важливий принцип для організації мікросервісної архітектури. Його головна ідея – це виділити по окремій базі даних під кожний мікросервіс.

Варто зауважити, що в моноліті також може мати місце децентралізації, на рівні серверного коду. Але в більшості випадків на рівні бази даних ця ізольованість відсутня, що після довгого процесу розробки веде до значного ускладнення сутностей та зв’язків між ними.

Тож, щоб позбавитися цієї проблеми, в мікросервісах загальна база даних відсутня, як показано на рисунку 1.4.

Цей підхід гарантує не лише ізольованість, а й спрощує реалізацію шаблону Polyglot Persistence, суть якого у виділенні окремої бази під певну потребу. Цьому можна слідувати і при використанні не мікросервісної архітектури, але це не є необхідністю.

Дотримання цього принципу разом із перевагами несе і деякі складнощі. Адже багато баз породжують багато контекстів, роботу з якими слід узгоджувати. Як вихід з положення можна було б використовувати техніку розподілених транзакцій. Але синхронна робота мікросервісів іде в розріз з основними принципами їх організації.

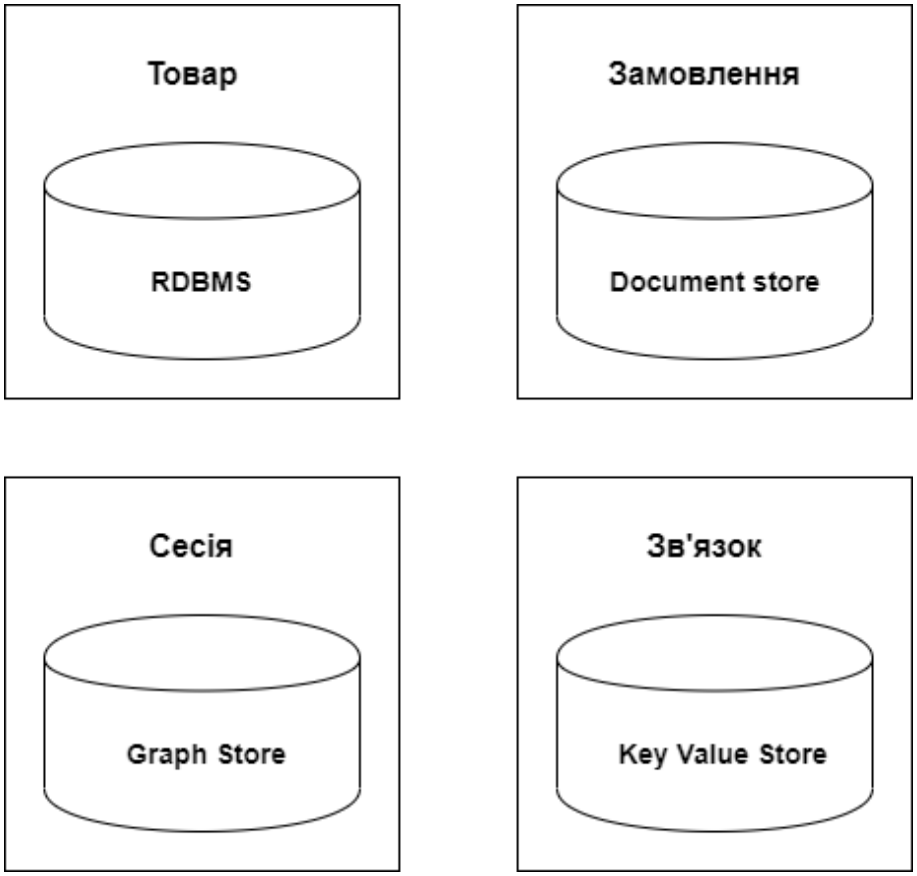


Рис 1.4 Приклад організації мікросервісів з окремими базами даних

Як рішення даної проблеми використовують відмову від постійної узгодженості даних, тобто використовують модель Eventual consistency (укр. Узгодженість в кінцевому рахунку)[2].

Узгодженість в кінцевому рахунку (англ. Eventual consistency) - одна з моделей узгодженості, яка використовується в розподілених системах для досягнення високої доступності, в рамках якої гарантується, що у відсутності змін даних, через якийсь проміжок часу після останнього оновлення («в кінцевому рахунку») всі запити будуть повертати останнім оновлене значення. Приклад оновлення запису зображено на Рис 1.5.

Такий підхід допускає, що не у всіх випадках є необхідність узгоджувати данні одразу після завершення транзакції. Тоді більшу розподілену транзакцію можна розбити на декілька менших локальних. Як обробляти неузгодженість даних є деталями імплементації і залежить від

конкретних бізнес потреб. В якомусь випадку можна не брати до увагу тимчасову неузгодженість, в іншому використовувати моніторинг або більш складне архітектурне рішення. Якщо використання цього підходу не є можливим, то як останній вихід можна використати розподілені транзакції.

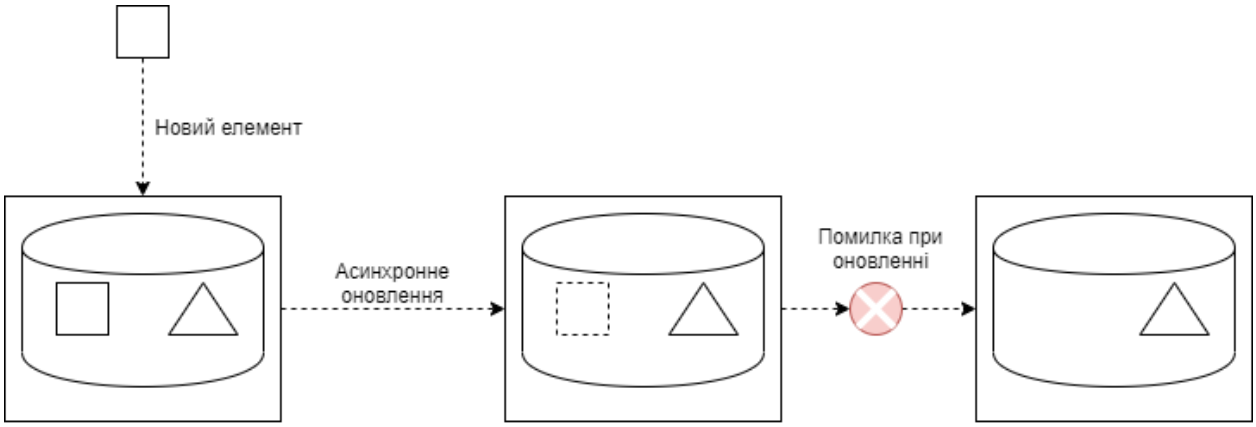


Рис 1.5 Приклад оновлення запису при використанні Eventual consistency

1.1.7. Необхідність автоматизації в процесах розробки та ітеративний розвиток

Мікросервісна система може складатися як з декількох так і сотень сервісів, які вимагають концептуально різних середовищ. При такій складності розгортання на сервері може стати дуже складним і довготривалим процесом. Тому розробка і підтримка мають бути автоматизовані.

Цього можна досягти використовуючи спеціальні технології CI\CD (Continuous Integration \ Continuous Delivery). Вони дозволяють налаштувати різні стадії та автоматизувати процес розгортки системи.

Для надання різного середовища сервісам найпопулярнішим засобом є Docker, що дозволяє будувати та запускати контейнери для сервісів незалежно від платформи, а в купі з Kubernetes ще й зручно розгортати цілу систему на кластері чи розподілених машинах.

Ітеративність розвитку мікросервісу – це підхід до розробки при якому в кінці кожної ітерації на виході є робочий продукт, але з неповною

функціональністю, яка додається з кожною новою ітерацією. Тобто розробка виконується за Agile методологією. При такій розробці використання CI/CD є необхідністю для організації ефективного процесу розвитку. До того ж кожний мікросервіс розвивається незалежно від інших і це може відбуватися в різних командах.

1.2.Доцільність мікросервісів у порівнянні з монолітом

Мікросервісний підхід до організації системи несе чимало проблем. Найперші з них, що можуть з'явитися – це організаційні питання. Наприклад, як керувати системою, так щоб вона залишилася в робочому стані, коли існує сотні мікросервісів, які часто змінюють свій стан? Яка команда буде відповідальна за інтеграційні тести та як писати їх для такого великого скупчення елементів? Як відслідковувати помилки та хто є відповідальним за них?

Ці та інші можливі проблеми по'вязані з організацією є дуже серйозними і можуть значно погіршити процес розробки. Рішенням є використання гнучкої методології Agile разом з різними DevOps технологіями.

Також є архітектурні питання. Як перейти від моноліту з синхронними подіями та єдністю даних до мікросервісної розподіленої системи з багатьма дрібними елементами, в якій до того ж може мати місце невідповідність даних. Ця проблема змушує задуматися архітекторів про доцільність зміни монолітної організації на мікросервісну.

Тобто варто сказати, що, якщо моноліт в робочому стані, не має серйозних проблем і не має необхідності в його розширенні в сторону розподіленої системи, то і не треба змінювати архітектуру, адже це дуже трудомісткий процес.

Але якщо все ж таки проблеми в моноліті присутні, то мікросервіси можуть бути виходом із ситуації.

Виокремлення окремих незалежних компонентів має багато переваг: простота контексту, гнучкість розвитку, легке масштабування та керування. Незалежність та невеликий розмір дають ряд переваг. Адже можна відмовитися від дорогої машини, що брала на себе усі функції. В більшості випадків мікросервіси можна розгортати на дешевих машинах, бо вони мають дуже обмежений функціонал. На практиці багато маленьких машин будуть не лише коштувати менше, а ще й працювати більш ефективно за дорогу машину призначену для монолітної системи.

Тому дуже розповсюдженим, хоча і не обов'язковим, є підхід при якому на кожний мікросервіс виділяється по окремому серверу.

Також як явний плюс для мікросервісної архітектури може слугувати те, що лідери індустрії такі як Google, Netflix та багато інших, чиї системи побудовані на мікросервісах, показують приголомшені результати. Їх продукти є дуже гнучкими і це дозволяє додавати нові неймовірно швидко у порівнянні з конкурентами.

В той же час існують компанії, які зручно відчувають себе при використанні моноліту. Наприклад, Amazon працював на моноліті, навіть коли був вже гігантом на ринку. Сайт для газети Guardian базується на мікросервісах навколо моноліту. Тобто підхід до рішення тих чи інших задач залежить насамперед від самих задач, тому можуть виникати і гібридні системи, що поєднують у собі декілька підходів.

Висновки до розділу 1

Мікросервіси – поширене у наш час архітектурне рішення. Воно дозволяє створювати гнучкі розподілені системи з незалежними компонентами, що має як свої плюси так і мінуси. Серед плюсів можна назвати гнучкість розробки окремих компонентів, більш дешеві у порівнянні з монолітом сервера, незалежність компонентів між собою та легкість додання нових за потреби. Серед мінусів можна виділити асинхронну природу, складну систему організації зв'язку, що вимагає досвідчену команду, проблеми з неузгодженістю даних.

Було розглянуто основні принципи, яким має слідувати мікросервісна система, серед них невеликий розмір, незалежність, обмежений контекст, шаблон Smart endpoints and dumb pipes, підхід Design for Failure, децентралізація, автоматизація в процесах розробки та підтримки та ітераційний розвиток.

Показано, в яких випадках є доцільним переходити з моноліту на мікросервіси та наведено плюси та мінуси обох підходів. Також були наведені приклади вдалої імплементації цих архітектур гігантами цифрової індустрії.

РОЗДІЛ 2
АНАЛІЗ ЗАСОБІВ ОРГАНІЗАЦІЇ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

Для організації запропонованої (мікросервісної) архітектури використання певних технологій є необхідністю. Для обміну повідомленнями необхідний посередник (брокер), що підтримує мікросервіси в незалежному один від одного стані і в той же час дозволяє взаємодіяти та виконувати спільну роботу. Також для розробки, розгортки та керування життєвим циклом необхідна система, що надає такий функціонал.

2.1. Брокер повідомлень

Брокер повідомлень буде виступати посередником в розроблюваній системі. Тому слід дослідити природу їх роботи, основні можливості, поняття та популярні системи, що існують на теперішній час.

2.1.1 Основні поняття брокерів повідомлень

Для того аби два сервіси (застосунки) могли обмінюватися повідомленнями між собою, необхідно спочатку вирішити який інтерфейс буде використаний для цього. В інтерфейс входять транспорт або протокол (SMTP, HTTP) та формат повідомлень, які будуть пересилатися через систему обміну. Формат може бути як строгий, наприклад, схема XML з певними вимогами до виду повідомлень, або ж узгодженість між розробниками, наприклад, про те, що всі повідомлення мають мати поле ідентифікатор. Поки є дотримання контракту інтерфейсу сервіси, що виступали учасниками обміну в системі, можуть бути зміненими, адже вони є незалежними.

Як правило, в системі обміну повідомленнями є спеціальний посередник між двома системами. Це допомагає ще більше зменшити зв'язність між відправниками та отримувачами. Такими посередниками є брокери.

В своїй суті брокер повідомлень – це програма, яка транслює повідомлення з формального протоколу обміну повідомлень відправника в

протокол отримувача. Тобто можна сказати, що брокер слугує проміжною ланкою між відправником та отримувачем (рисунок 2.1).

Брокери повідомлень слугують як елементи комп'ютерних та телекомунікаційних систем, в яких є необхідність у спілкуванні між частинами системи використовуючи формально визначені повідомлення.

Брокер повідомлень є архітектурним шаблоном, що об'єднує в собі функціонал для трансформації та маршрутизації повідомлень.

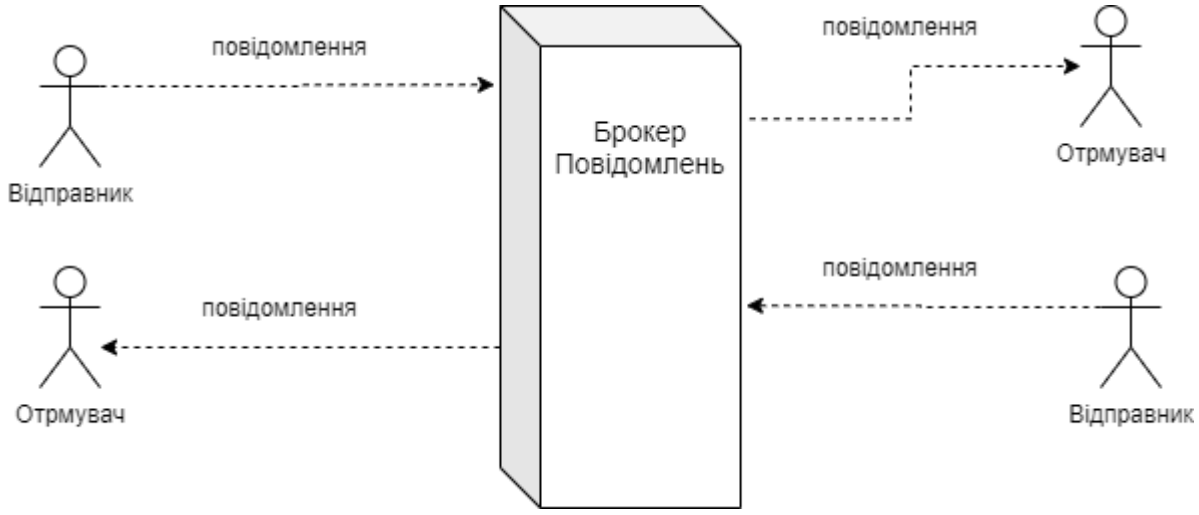


Рис. 2.1 Брокер повідомлень як проміжна ланка між відправником та отримувачем

Термін отримувач (або споживач) може мати різне значення в залежності від контексту. Але загалом отримувач в контексті обміну повідомленнями – це деякий застосунок, що споживає повідомлення. З іншого боку в протоколах обміну повідомлень є поняття підписки на доставку повідомлень. Тому ще виділяють підписку як термін для опису отримувача. Тоді споживача можна описати як підписку на доставку повідомлень, яка повинна бути зареєстрована на початку процесу обміну повідомленнями та в будь-який момент може бути скасованою.

Відправник в контексті обміну повідомленнями – це деякий застосунок, що створює повідомлення та відправляє їх, не маючи жодної інформації щодо підписок споживачів.

2.1.2. Області застосування брокерів повідомлень

Брокери повідомлень є гарним рішенням в системах з великою кількістю повідомлень. Проте задля інкапсуляції обміну часто використовуються і в системах з невеликим обігом. Брокер виступає як централізоване місце зберігання та обробки повідомлень, даючи змогу іншим сервісам або користувачем спілкуватися.

Брокери можуть бути дуже корисними при підключенні вже існуючого застосунку до системи. Наприклад, коли компанія хоче розширити функціонал своєї системи за рахунок підключення окремого застосунку. Одним з варіантів є перепис кодової бази нової частини, зміни контрактів бази даних, що звісно є дуже довготривалим і виснажливим для команди завданням. Альтернативний же спосіб з використанням брокеру – це підключити новий застосунок до централізованого обробника подій і навчити існуючу систему реагувати на події. В такому разі доведеться лише додати функціонал для цих подій, що набагато менш трудомістким, ніж повний перепис коду.

Як інший приклад використання можна навести IoT (Internet of Things) систему, що розрахована на велику кількість пристроїв, які мають постійно обмінюватися повідомленнями між собою. Брокер може виступити як дуже зручний і оптимізований спосіб реалізації цього випадку.

Можна виділити 4 головні функції, які виконують брокери повідомлень:

1) розділення відправника та отримувача;
2) зберігання повідомлень;
3) маршрутизація повідомлень;
4) перевірка і організація повідомлень.

В процесі обміну повідомлень існують два головні підходи: черга та публікація-підписка. Обидва зі своїми недоліками та перевагами. Підхід публікація-підписка дозволяє направляти повідомлення багатьом слухачам,

але є погано масштабованим. А черга ж дозволяє відсилати повідомлення лише одному споживачу, зате є гарно масштабованою.

2.1.3. Огляд існуючих брокерів повідомлень

Брокери повідомлень здобули популярність через свої переваги вже досить давно, тому зараз є багато різноманітних варіантів для вибору, наприклад, Kafka, Azure Event Hub, Service Bus, JMS, Redis, RabbitMQ та інші. Розглянемо деякі найпопулярніші з них.

2.1.3.1 Apache Kafka

Apache Kafka – брокер повідомлень написаний під Java платформу. Ця система була спроектована компанією Linkedin в 2010 році для власних потреб, так як існуючі в той час брокери не задовольняли потреб, бо були не достатньо швидкі і при цьому доволі важкими в плані споживання ресурсів. Через це розробники компанії спроектували масштабовану та відмово стійку систему, яка скоро виросла в повноцінну платформу обробки подій.

Технологія здобула популярність завдяки своїй сумісності, яка дозволяє використовувати Кафку з великим різноманіттям систем, серед яких веб застосунки, мікросервіси, системи моніторингу та аналітики, бази даних, розподілені системи обчислень та інші. З її допомогою можна створювати та керувати дуже складним системами.

Apache Kafka містить такий функціонал:

1) публікація (виробники) записів та підписка (споживачі) на потоки даних з гарною масштабованістю;
2) надійне зберігання потоків та розподілення даних по вершинах;
3) обробка потоків, коли вони прибувають та виконання різних маніпуляцій з ними, таких як об’єднання, агрегації та інших.

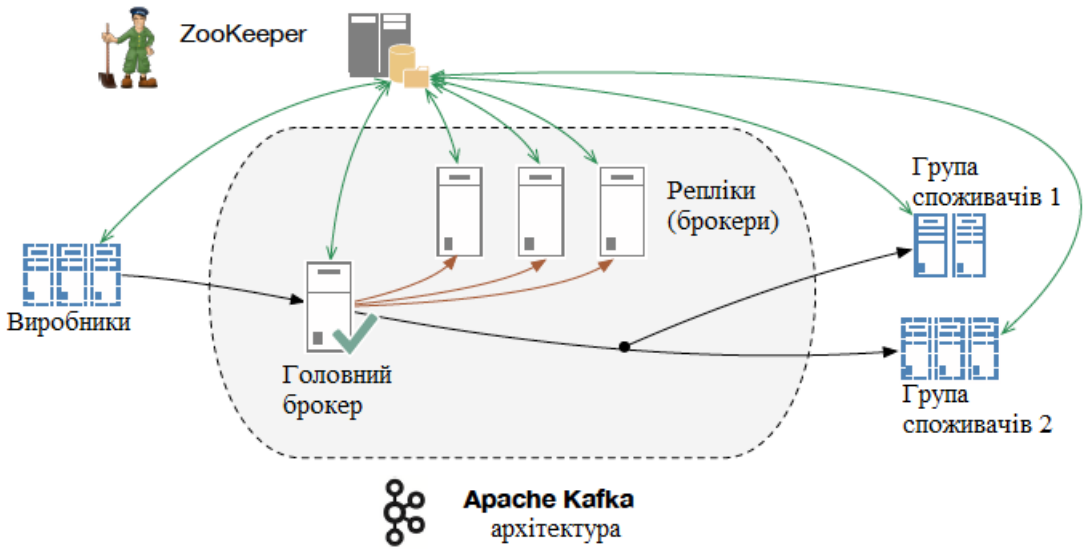


Рис 2.2 Архітектура Apache Kafka

На рисунку 2.2 зображено архітектуру Apache Kafka. Як видно виробники(відправники) та споживачі(отримувачі) отримують спілкуються між собою через систему брокерів з головним, який слугує як керівник, і репліками, що призначені для розподіленої обробки повідомлень. Для менеджменту всією розподіленою системою використовується Apache ZooKeeper.

Apache Kafka працює на логованій системі даних. Де лог (анг. log) - це упорядкована за часом, послідовність записів даних. Також в цій системі існують такі концепти як топіки (анг. topics) – впорядковані потоки записів, записи, які складаються з ключа, значення, та часу створення, та набір API, серед них Producer API (укр. АПІ виробників), Consumer API (укр. АПІ споживачів), Streams API(укр. АПІ потоків), Connector API(укр. АПІ з'єднувача).

Протокол TCP використовується для комунікацій серверів системи з клієнтами. Цей прокол легкий у використанні, ефективний та платформо незалежний, що дозволяє писати клієнтів на будь-якій мові чи платформі.

Однією з головних переваг Apache Kafka є те, що тут поєднуються два підходи: чергу і публікацію-підписку, і бере переваги з обох, що дає явний виграш над іншими традиційними брокерами.

До переваг Kafka можна віднести:

1) легкий старт;
2) потужність в обробці потоків записів;
3) надійність і відмовостійкість;
4) масштабованість;
5) безкоштовний продукт, що підтримується спільнотою;
6) гарно підходить для обробки в режими реального часу;
7) зручність інтеграції з Big Data технологіями.

Серед недоліків можна виділити такі:

1) недостатня кількість готових до використання елементів;
2) нема доцільної для таких об'ємів даних системи моніторингу;
3) залежність на сервіс Apache Zookeeper;
4) відсутність маршрутизації;
5) можливі проблеми при значному збільшенні кількості повідомлень.

2.1.5.2.1 RabbitMQ огляд

RabbitMQ є брокером повідомлень написаним на мові Erlang[6]. Завдяки тому, що це функціональна мова програмування, в якій заготовлено багато готових до використання в різних сферах компонентів, робота з цим брокером майже не вимагає ручних налаштувань. RabbitMQ є представником традиційних брокерів, на відміну від вище згаданого Kafka. Ця система знайшла своє місце як в невеликих стартапах, так і в громіздких та складних проектах. В основі RabbitMQ лежить Open Telecom Platform для мови Erlang. Він надає можливості для написання логіки кластеризації та обробки відмов. Для цього брокеру написані спеціальні бібліотеки під всі популярні мови, що можуть використовуватися для розробки систем, які вимагають обміну повідомленнями. Тому процес розробки значно полегшується.

На сьогодні RabbitMQ – це один з найстарших брокерів, що був створений в 2007 році компанією Rabbit Technologies Ltd. як імплементація AMQP. AMQP – це відкритий протокол для обміну повідомлення зі складними видами маршрутизації. Він дозволяє використовувати брокери повідомлень поза межами Java середовища. RabbitMQ є дуже гарним рішенням у використанні з .NET, Java, Python, PHP, JavaScript, Ruby, Go та іншими мовами разом із фреймворками до них. Тож, головною перевагою цього брокеру можна назвати велику кількість готових плагінів і бібліотек.

В основі RabbitMQ, який був спроектований для загального користування, лежить принцип обміну повідомленнями публікація-підписка[5]. В залежності від потреби процес обміну повідомленнями може бути синхронними або асинхронним. Як головні особливості цього брокеру можна виділити:

1) підтримка різних протоколів і видів обробки черг повідомлень, легко змінювана маршрутизація по чергам, різні види обміну;
2) кластеризування розгортання брокеру;
3) легка інтеграція з системами автоматизації побудови та розгортки, наприклад, Docker;
4) підтримка різних сучасних мов програмування;
5) готові до використання авторизація та аутенфікація.

До переваг можна віднести:

1) підходить для різних мов програмування та протоколів спілкування;
2) можна використовувати на різних операційних системах та хмарних рішеннях;
3) простий початок та розгортка;
4) містить багато готових корисних інструментів;
5) наявність системи моніторингу в режимі реального часу;
6) гарно масштабується;
7) може обробляти більше 500 000 повідомлень в секунду.

До недоліків:

1) потребує, щоб Erlang був встановлений та запущений на сервері;
2) не багато можно налаштувати через мову програмування;
3) можуть виникати проблеми в роботі з великою кількістю повідомлень.

2.1.5.2.2 RabbitMQ основний функціонал та терміни

На рисунку 2.3 зображено взаємодію виробників і споживачів через брокер RabbitMQ. З нього видно що в цій системі існують поняття exchange(укр. обмін) та queue (укр. черга), які прийшли як імплементація моделі AMQP 0-9-1[16].

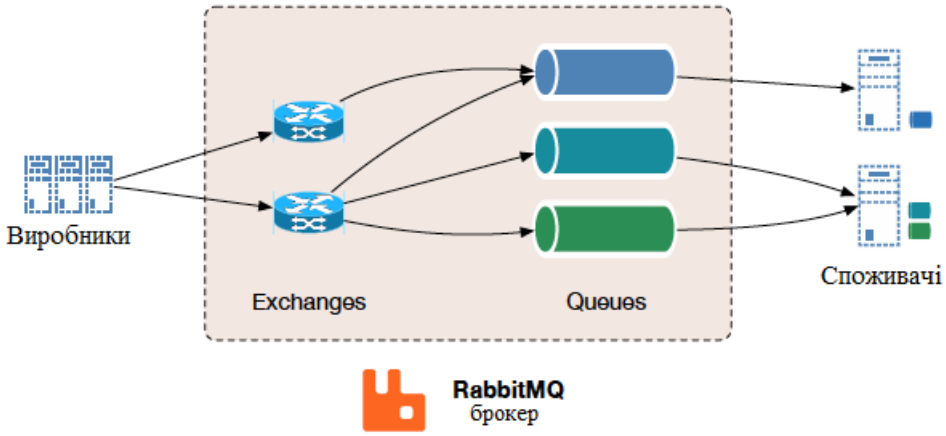


Рис. 2.3 Базова архітектура брокеру RabbitMQ

Exchange - агент обміну, що слугує для маршрутизації повідомлень в середовищі RabbitMQ за допомогою спеціальних атрибутів, таких як заголовки (анг. headers), прив'язок (анг. bindings) та ключів маршрутизації (анг. routing key).

Binding – це своєрідний зв'язок між агентом обміну та чергою, який встановлюється при створенні черги.

Routing key – атрибут повідомлення, який встановлюється відправником, який агент обміну порівнює за допомогою певних правил з binding черги (залежно від його виду) та направляє в підходящу чергу.

AMQP 0-9-1 (Advanced Message Queuing Protocol) – протокол який дозволяє клієнтським застосункам спілкуватися з проміжним брокером повідомлень. Ця модель працює за наступним принципом: повідомлення надсилаються не напряму в черги, а спочатку в обрану exchange, яка вже залежно від виду направляє повідомлення в ту чи іншу чергу.

Шлях повідомлення через систему з RabbitMQ можна зобразити як на рисунку 2.4.:

1) виробник надсилає повідомлення в exchange;
2) повідомлення потрапляє в exchange, до якої переходить право вирішення, до якої черги повідомлення буде направлено;
3) для того, щоб повідомлення потрапило в чергу, черга повинна бути прив’язана до exchange. Якщо існують прив’язані черги, то exchange направляє повідомлення в них;
4) повідомлення будуть залишатися в черзі поки отримувач не обробить їх;
5) споживач отримує повідомлення і оброблює його.

В RabbitMQ є також поняття acknowledgement (укр. підтвердження)[16]. Кожен отримувач в певний момент часу після прийому повідомлення може надіслати сигнал підтвердження, якщо ж цього не відбудеться, то повідомлення повернеться в чергу або буде перенаправлене в спеціальну чергу для повідомлень, що не були підтверджені.

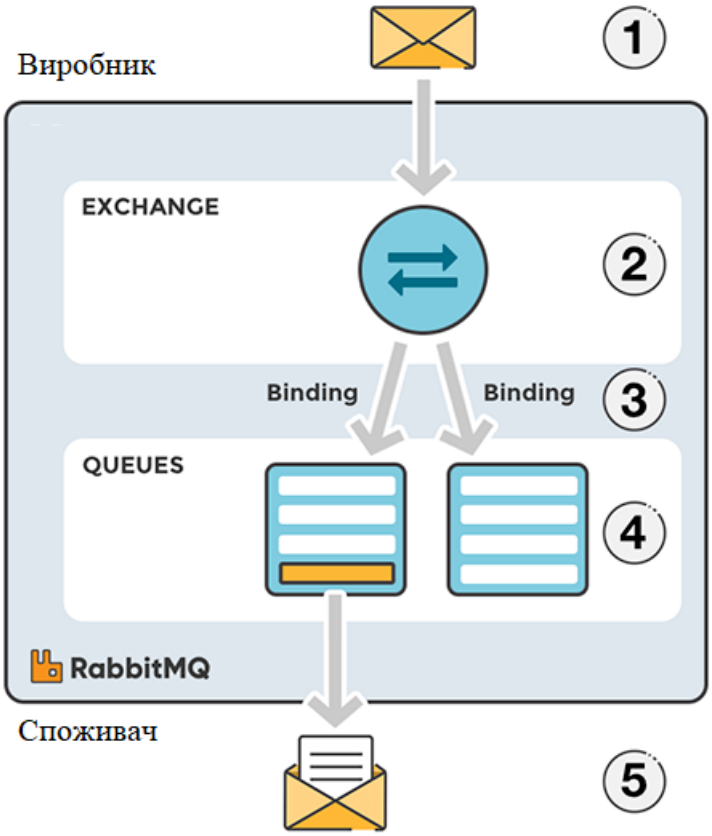


Рис. 2.4 Шлях повідомлення в системі RabbitMQ

2.1.5.2.3 Види Exchange в RabbitMQ

Існують такі види exchange в RabbitMQ:

1) Direct Exchange;
2) Default Exchange;
3) Topic Exchange;
4) Fanout exchange;
5) Headers Exchange;
6) Dead Letter Exchange.

Direct Exchange направляє повідомлення базуючись на routing key повідомлення. Правило для обирання черги: повідомлення направляється до черги, в якої binding key повністю ідентичний routing key. Наприклад, якщо binding key черги ‘key1’ і routing key повідомлення ‘key1’, то воно буде направлене в цю чергу, як зображено на рисунку 2.5.

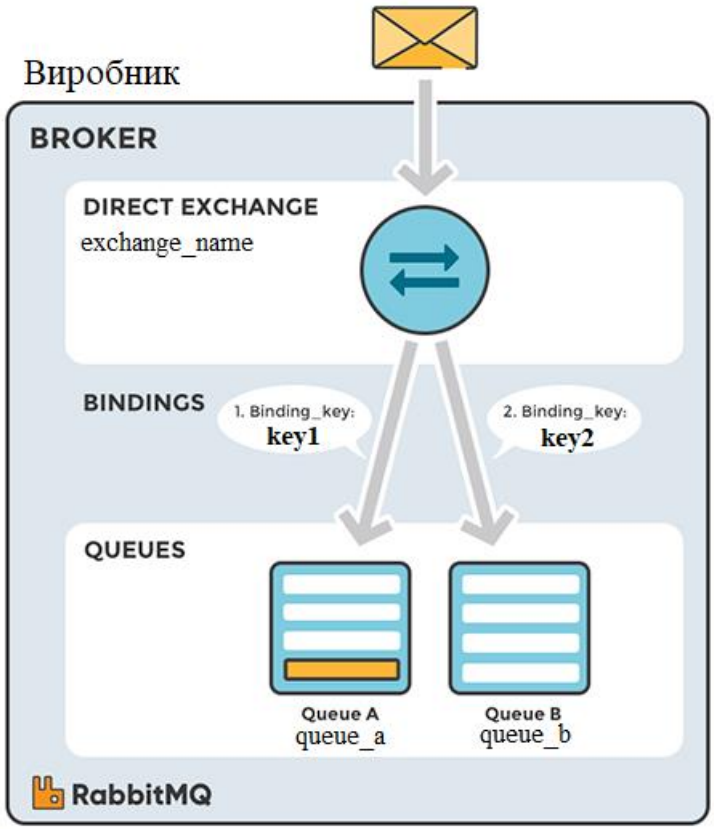


Рис. 2.5 Direct Exchange

Default Exchange – це агент обміну за замовчуванням, який за фактом є direct exchange з пустим іменем (“”). Кожна черга автоматично зв’язується з цією exchange за допомогою binding key, який такий самий як і ім’я черги . Для того щоб надіслати повідомлення в певну чергу через цей exchange необхідно вказати ім’я черги як routing key.

Topic Exchange направляє повідомлення базуючись на частковому співпаданні за шаблонами між routing key повідомлення та binding key черги. В такому випадку повідомлення може надійти в декілька черг, якщо є співпадання. Routing key має виглядати, як набір слів, що розділені крапкою, наприклад, country.eu.ukraine. Також замість слів може використовуватися зірочка (*), щоб вказати лише бажані слова (country.*.ukraine.* або *.*.ukraine.*). Ще може використовуватися решітка (#), щоб вказати на 0 або більше слів замість неї (country.eu.ukraine.#). Формат binding key може бути таким самим, як і в routing key.

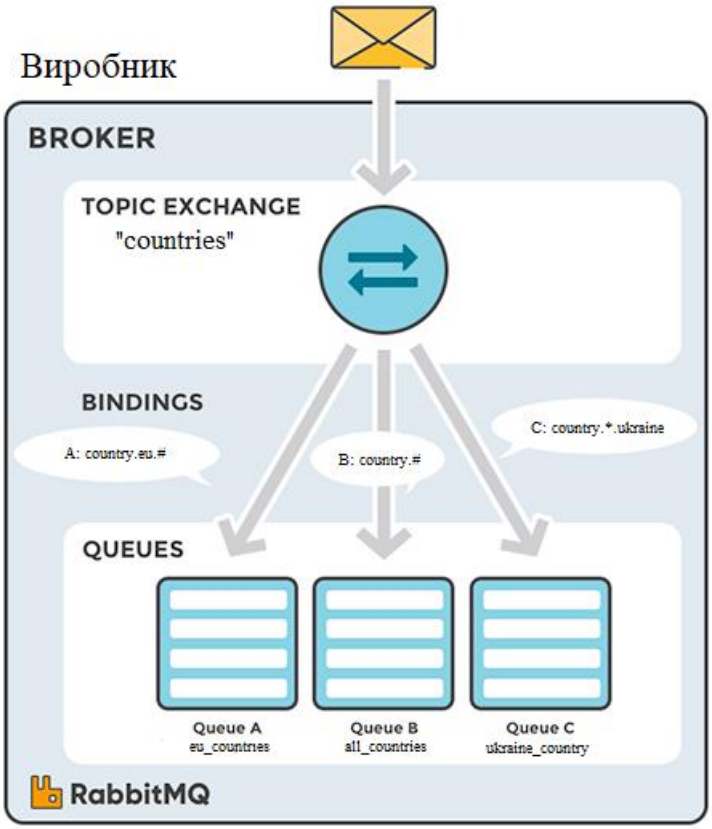


Рис. 2.6 Topic Exchange

На рисунку 2.6 зображено приклад Topic Exchange. Як приклад, для того щоб відправити повідомлення в чергу з назвою “eu_countries” підійдуть такі ключі: “country.eu.#”, “country.eu.ukraine.city”, “country.eu” та інші. А в чергу “ukraine_country” : “country.*.ukraine”, “country.eu.ukraine”, “country.as.ukraine” та інші.

Fanout Exchange відправляє вхідне повідомлення до всіх черг, які прив’язанні до неї, не дивлячись на routing key. Навіть якщо ключ буде переданий, то він буде проігнорований. Приклад зображено на рисунку 2.7.

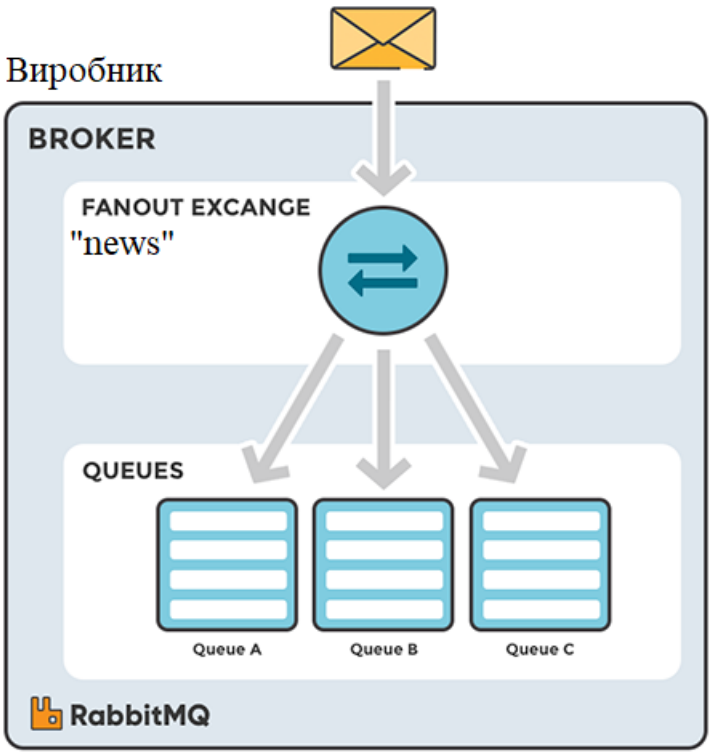


Рис 2.7 Fanout Exchange

Headers Exchange відправляє вхідне повідомлення, порівнюючи передані заголовки (headers) і їх опціональні значення. Цей вид схожий на Topic Exchange, з відміною в тому, що порівнюються headers, а не routing key. За допомогою поля “x-match”, що додається за такого обміну, можна налаштовувати, як саме заголовки будуть порівнюватися. “x-match” може мати два допустимих значення. Якщо необхідно, щоб хоча б один заголовок у повідомлення та прив’язці співпадав, то слід обрати значення “any”, якщо ж треба зробити, щоб всі заголовки співпадали, то “all” (це значення також використовується за замовченням, якщо параметр x-match не було надано).

На рисунку 2.8 зображено приклад Headers Exchange. В такому випадку повідомлення 1 (Message 1) буде доставлено в чергу A (Queue A). Повідомлення 2 (Message 2) в Queue A та Queue B. А повідомлення 3 (Message 3) не буде відправлено до черг, адже нема відповідності.

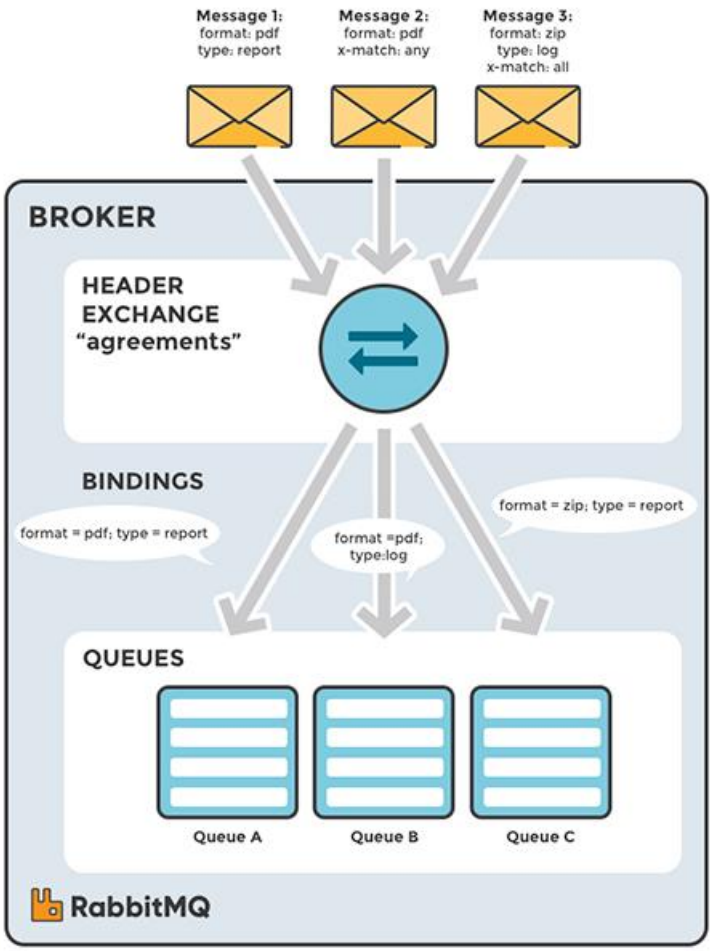


Рис 2.8 Headers Exchange

Dead Letter Exchange – спеціальний вид exchange, куди потрапляють повідомлення, які не можуть бути доставлені в черги.

2.2. Docker як інструмент для автоматизації побудови мікросервісів

В наш час існує багато різних середовищ, в яких працюють застосунки. Для складних систем, таких як мікросервісні, в яких кожна частина може працювати на окремій машині з відмінним від інших середовищем та мати декілька реплік на інших машинах використання спеціальних інструментів для керування процесами розробки і розгортки є необхідністю. Найпопулярнішим таким інструментом є Docker.

2.2.1. Передумови для використання Docker

Зазвичай великі проекти містять багато налаштувань. Вони розробляються в певному середовищі, яке може мати багато залежностей та зв'язків, наприклад система працює в Linux операційній системі зі специфічними для застосунку змінними середовища та потребує, щоб спеціальні стороннє програмне забезпечення було встановлено та запущено. Все це необхідно враховувати і відповідно налаштовувати при розгортці на сервері. Це довготривалий та складний процес, адже доводиться працювати з віддаленим сервером, який ще й може не відповідати вимогам[7].

А коли в системі не один сервіс, а багато з повністю різними залежностями та операційними системами, доводиться цей процес пророблювати для кожного з них. При ітераційній розробці, коли перезапуск сервісів може відбуватися кожен день, цей підхід не влаштовує. Адже майже всі ресурси підуть на запуск та виключення серверів.

Це можна було б вирішити за допомогою написання спеціальних скриптів. Але вони не будуть платформо незалежними, і в разі зміни бібліотек чи операційних систем доведеться переписувати їх повністю.

Тому доцільно використовувати існуюче програмне забезпечення для автоматизації побудови і розгортки, такі як Docker.

2.2.2. Огляд Docker та основні поняття

Docker – це відкрита платформа для автоматизації побудови, розгортки та керування сервісами з підтримкою контейнеризації. Ця система почала свій розвиток як надбудова на LXC(Linux containers).

LXC – це своєрідний метод віртуалізації на рівні операційної системи, яка призначена для того, щоб запускати багато контейнерів в середовищі однієї системи, які до того ж ізольовані один від одного[7].

В докер системі є два основних поняття: image та container (укр. контейнер).

Docker image (укр. докер образ) – це шаблон, що містить в собі набір програм, утиліт, які разом формують образ машину з середовищем готовим до використання. Наприклад образом може бути операційна система Windows з встановленою на ній .NET платформою.

Docker Container (укр. докер контейнер) – це обгортка над певним докер образом, що дозволяє налаштовувати та запускати його.

В контексті мікросервісів кожен контейнер буде містити в собі одну вершину.

Тобто можна провести аналогію, що образ це набір певних програм для машини, а контейнер вже є самою машиною.

Docker складається з трьох основних частин:

1) docker daemon;
2) консольна програма docker;
3) Docker Registry (реєстр).

Docker daemon можна вважати ядром docker системи. Це спеціальний демон процес, працюючий на машині, що є хостом для докеру, який відповідальний за завантаження образів, запускати контейнери, моніторити стан запущених контейнерів, включаючи логування з контейнерів та налаштовувати мережу між різними контейнерами. Також саме ця частина створює образи під контейнери[8].

Docker (docker в консолі) – це консольна програма, що дозволяє керувати демоном процесу докер використовуючи HTTP. На відміну від docker daemon це проста утиліта, через що вона і працює доволі швидко. При збиранні нового образу роль цієї команди в тому, щоб передати архів з папкою в docker daemon, який далі виконує всю роботу, через це краще збирати великі образі на локальній машині, щоб не пересилати великі архіви через мережу[18].

Деякі з основних команд docker:

1) docker pull – завантажити образ;

2) docker build – побудувати образ;

3) docker run – запустити контейнер;

4) docker stop – зупинити контейнер.

Registry (реєстр) – реєстр для зберігання образів. Docker Hub - це найбільший публічний реєстр, в якому зберігається безліч готових образів, це може бути образ з середовищем Java, .NET, бази даних MySql, брокера RabbitMQ та багато іншого. Також можна користуватися локальними чи відмінними від Docker Hub публічними реєстрами.

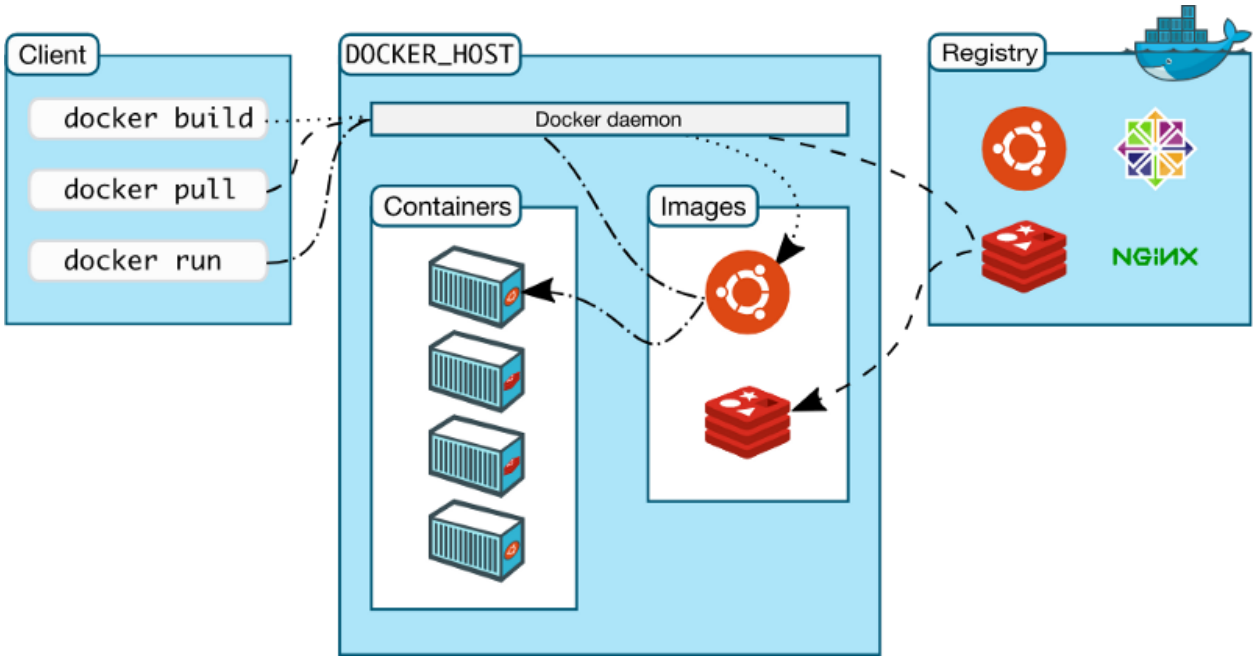


Рис 2.9 Взаємодія основних компонентів Docker

З рисунку 2.9 видно, що docker client, docker daemon та registry можуть знаходитися на різних хостах. Клієнтські команди слугують для делегації роботи докер процесу, а реєстр містить готові образи і в нього можна завантажувати власні.

2.2.2.1 Створення Docker Image та Docker Container

Образ можна створити власноруч або завантажити готовий з реєстру.

Для того аби створювати образи існує спеціальна мова, за допомогою якої описується інструкції. Ці інструкції прописуються у спеціальному файлі

образу під назвою Dockerfile (без розширень). Далі, знаходячись в папці з Dockerfile, можна запустити docker build команду для побудови образу. Також можна передати положення файлу за допомогою параметру –f, у випадку, якщо команда запускається поза папкою[18].

Образ формується з набору шарів. Кожній інструкції у файлі образу відповідає окремий шар. Шар представляє собою набір даних, шо відрізняються від шару, що йшов перед ним. Кожний шар будується на основі попереднього. Кожний шар образу доступний лише для читання.

При створенні контейнеру додається ще один шар, який вже є доступним для запису, його ще називають шаром контейнеру. Тут будуть зберігатися зміни зроблені в контейнері під час його роботи, такі як робота з файлами. На рисунку 3.2 зображено структуру шарів для контейнеру образу, Dockerfile якого складається з 4 інструкцій. Як видно з рисунку кожен шар слідує (з низу до гори) за попереднім і може мати різний розмір. Для цього рисунку Dockerfile міг би виглядати наступним чином:

```
FROM python:3.7-slim
COPY . /usr/app/src
WORKDIR /usr/app
CMD ["python", "-u","main.py"]
```

Для того аби не створювати образ системи з python використовуємо існуючу полегшену версію образу за допомогою інструкції FROM. Команда COPY слугує для копіювання між шарами. За допомогою WORKDIR змінюємо поточну директорію. CMD слугує, як точка входу в образ. Після запуску контейнеру за допомогою команди docker run з’являється ще один шар контейнеру, в якому будуть відслідковуватися зміни.

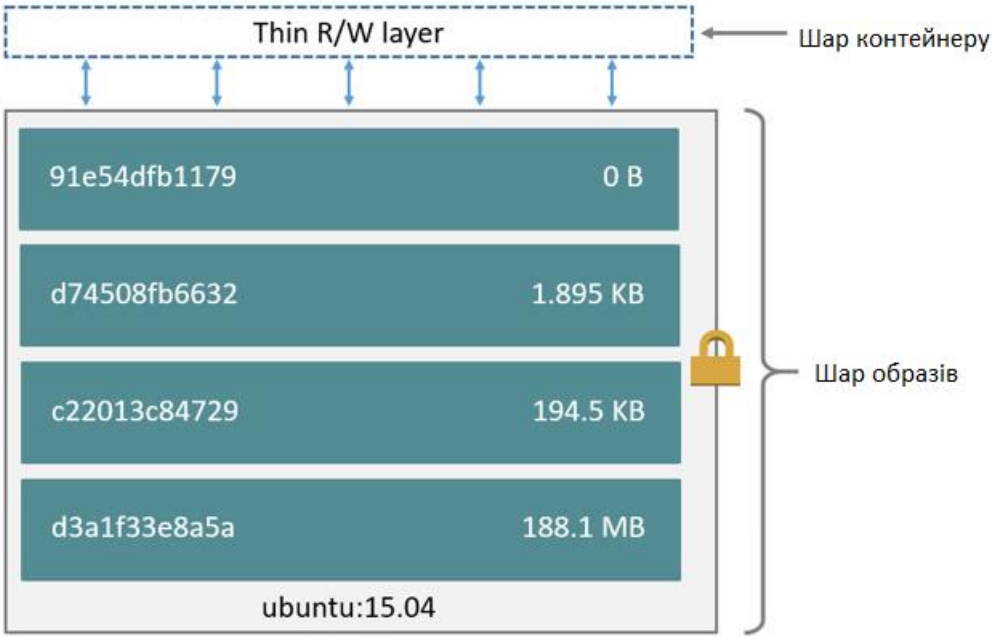


Рис. 3.2 Приклад організації шарів контейнеру для 4 інструкцій в Dockerfile

2.2.2.2 Docker Volume

Як було вказано вище шар контейнеру дозволяє зберігати роботу операцій з файловою системою, таких як видалення, редагування, створення файлів. Цей фунціонал підтримується за замовчуванням і не потребує додаткових налаштувань. Проте в такому підході є один суттєвий недолік: після знищення контейнеру всі дані зникнуть разом із шаром. Це є допустимо для тимчасових файлів, але для постійних необхідне інше рішення.

Цим рішенням є Docker Volume(укр. том докеру). Том – це спеціальна файлова система, яка знаходиться на машині поза контейнарами та слугує засіб зберігання даних між машиною, що хостить докер, та контейнерами. Проте їх керуванням займається саме докер, а не стороння програма на машині[18].

Основні властивості томів:

1) засіб для зберігання постійних фалів;
2) знаходяться поза контейнерами і незалежні від них;

3) ефективні запис та читання;
4) доступні для використання декількома контейнерами одночасно;
5) можуть бути іменованими;
6) зручні для тестування;
7) можна заздалегідь наповнити даними;
8) можна шифрувати.

В командній рядку можна працювати з томами (створювати, видаляти, змінювати та інше) за допомогою інструкції docker volume та наступних команд в залежності від потреб. Цими командами можуть бути create (створити том), prune (видалити не використовувані томи), ls (вивести список томів з корисною інформацією), rm (видалити том).

Для того аби прив'язати том до контейнеру потрібно при старті (docker container run) вказати в параметрі --mount том та папку, що буде йому відповідати в контейнері[18].

2.2.2.3 Docker Networking

Можливість з'єднувати сервіси разом – це саме те, що робить потужним інструментом для розробки. Docker Networking – це механізм для створення зв'язків між контейнерами. Існує 4 різних режими роботи:

1) bridge mode (укр. режим міст);
2) host mode (укр. режим хосту);
3) container mode (укр. режим контейнеру);
4) no networking (без мережі).

Приклад Bridge Mode зображено на рисунку 2.10. В режимі мосту докер процес (docker daemon) створює віртуальний Ethernet міст з назвою docker0, який автоматично пересилає всі пакети між всіма інтерфейсами мережі, що приєднанні до нього. Докер також під'єднує всі контейнери до цієї мережі за допомогою двох однорангових інтерфейсів (анг. peer interface). Перший з них буде eth0 інтерфейсом контейнеру, а другий буде знаходитися в просторі

імен хосту. Ще одне, що робить процес докеру, це призначення IP-адреси для мосту.

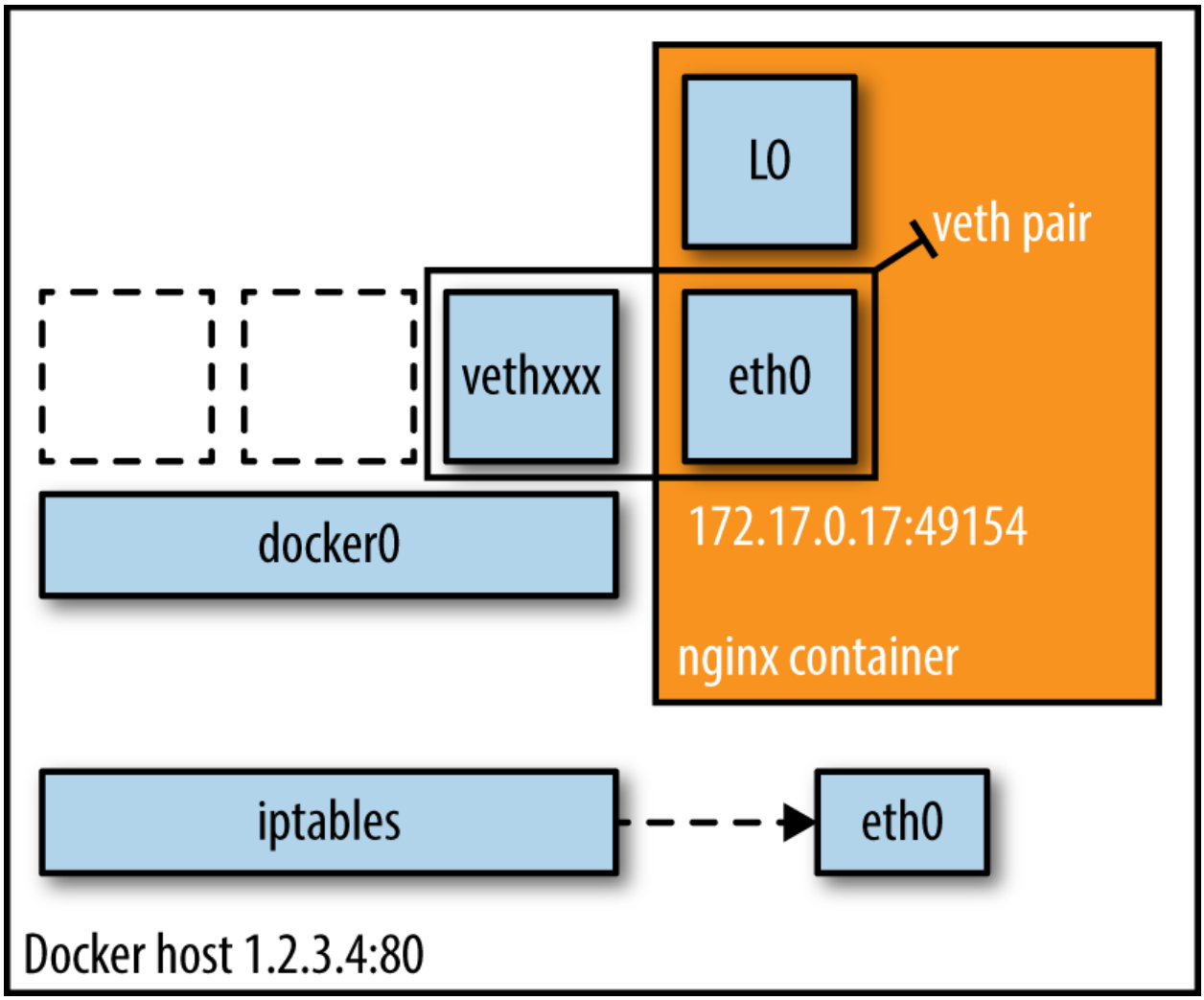


Рис. 2.10 Приклад Bridge Mode

На рисунку 2.11 зображено Host Mode, який нівелює ізоляцію контейнеру. Так як контейнер використовує простір імен машини, що є хостом, то контейнер буде доступний в публічній мережі. Як зображено на рисунку 2.11 ip хосту еквівалентний ip контейнеру. Цей режим є набагато швидшим за Bridge Mode, так як відсутня зайва маршрутизація, але він є менш безпечним, адже до контейнеру можна доступитися з зовнішньої мережі[18].

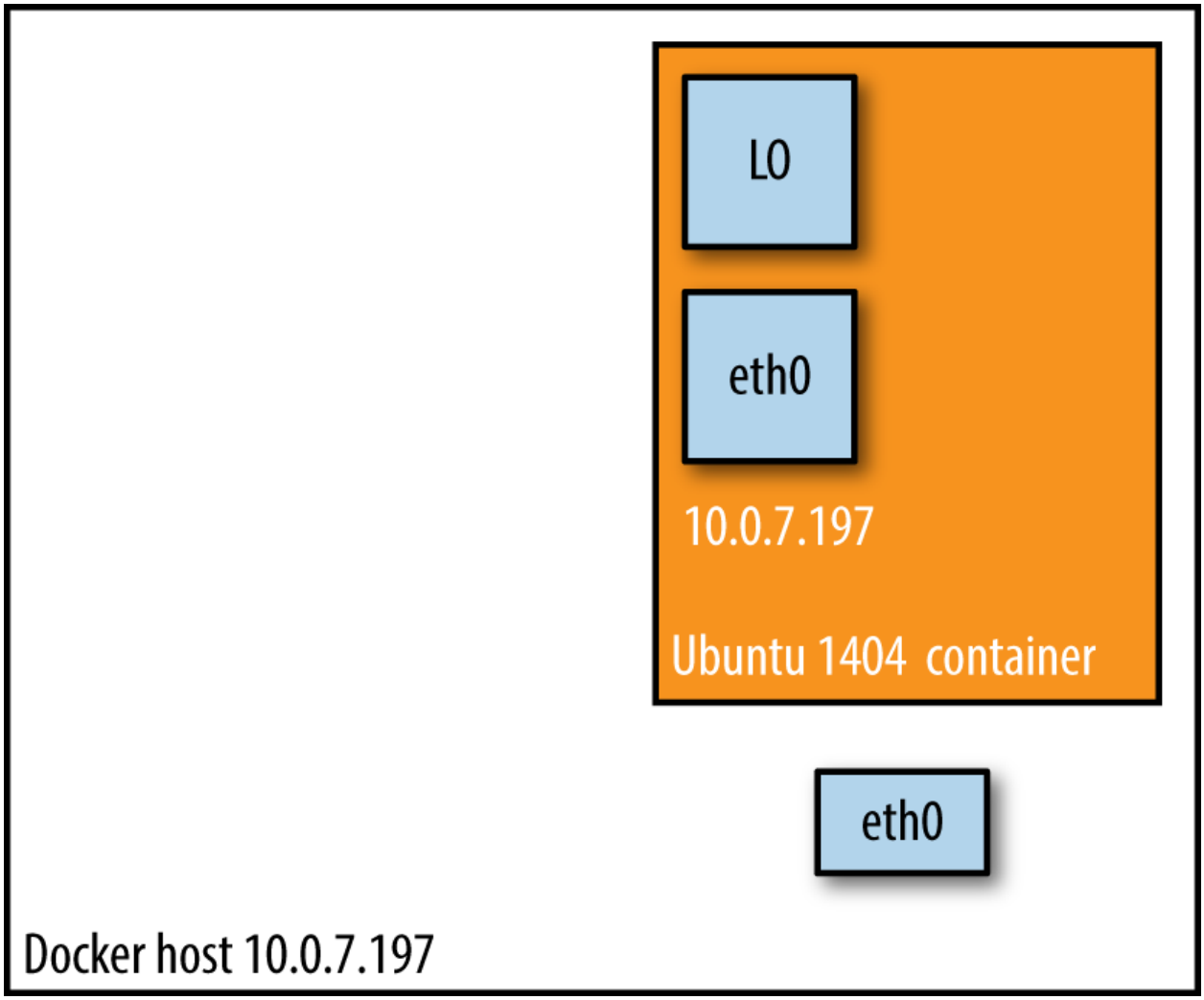


Рис 2.11 Приклад Host Mode

Container Mode надає функціонал для використання мережевого інтерфейсу іншого контейнеру. Цей механізм часто використовують для створення власних мережевих побудов.

Режим No Network, як слідує з назви, відключає мережу для контейнеру. Та насправді цей режим створює мережу для контейнеру але робить налаштувань. Це може стати в нагоді, якщо необхідно створити власну мережу або не підключати контейнер до мережі взагалі.

2.2.2.4 Docker Compose

Система, особливо мікросервісна, може складатися з багатьох сервісів. І для побудови та розгортки всіх їх необхідно написати команди консольної

утиліти для кожного з них, що є дуже не зручним. Та в Docker є вирішення цієї проблеми – Docker Compose.

Docker Compose – технологія, яка поставляється разом з Docker, яка призначена для налаштування та розгортки контейнерів. Вона надає той же функціонал, що і звичайний докер, але дозволяє запускати багато контейнерів однією командою (рис 2.12).

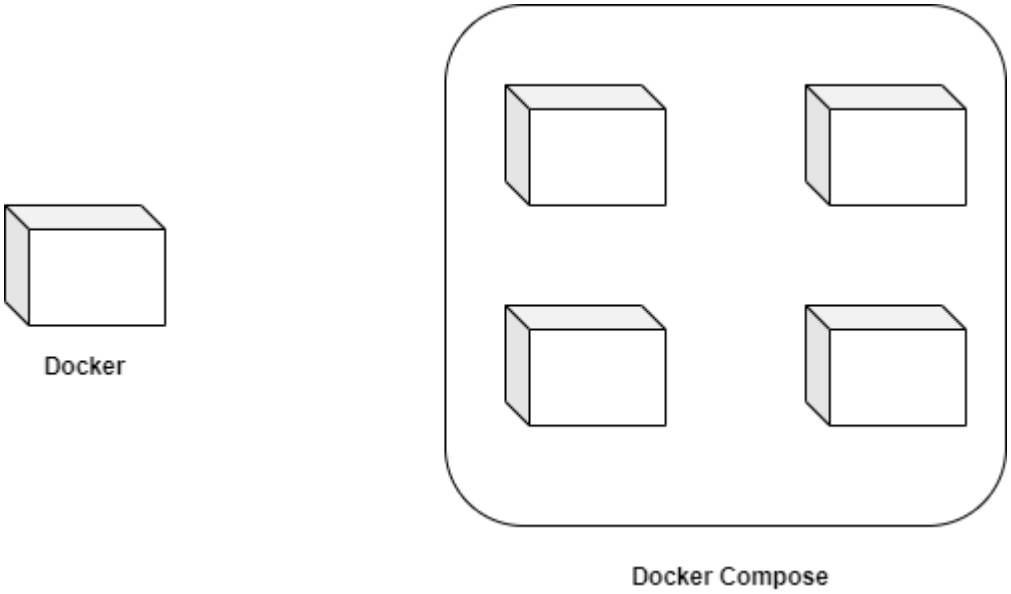


Рис. 2.12. Робота з багатьма контейнерами як перевага docker compose

Головними складовими Docker Compose є файл docker-compose.yml та утиліта docker-compose.

Файл docker-compose.yml пишеться на мові YAML[18]. В ньому можна написати конфігурацію для кожного сервісу. Серед основних можливостей, які надає цей файл можна виділити:

1) окремі налаштування для сервісів в одному місці;
2) приписування образу до сервісу;
3) вказування шляху до Dockerfile;
4) визначення ім’я контейнеру;
5) підключення томів;
6) просте з’єднування контейнерів у мережі;
7) призначення DNS адреси в середовищі докер для сервісу;

8) надання змінних середовища;

9) налаштування портів для публічного доступу;

10) створення томів;

11) перевірка працездатності.

Docker-compose утиліта дозволяє будувати та розгортати системи за допомогою конфігурації з файлу docker-compose.yml. Основними командами є:

1) build – побудова сервісів;

2) up – підняття (запуск) сервісів;

3) down – опущення (вимкнення) сервісів.

Ці команди також можна використовувати для роботи з одним або декількома сервісами, що представлені в файлі docker-compose.yml, передавши їх імена як параметри.

Висновки до розділу 2

В даному розділі було розглянуто брокери повідомлень, їх основні принципи та властивості. Як приклади найпопулярніших з них було наведено Apache Kafka та RabbitMQ.

Кожна з цих систем має як переваги так і недоліки. Apache Kafka є легшою і дозволяє обробляти більші потоки даних, а RabbitMQ має більш гнучку систему маршрутизації та багато вбудованих компонентів, що спрощують розробки. Саме через ці переваги, RabbitMQ буде використовуватися як брокер в системі, що буде створена в цій роботі. До того ж його пропускних здібностей достатньо для невеликого обсягу повідомлень.

Тому RabbitMQ був розглянутий ширше. А саме були описані принципи його роботи та види маршрутизації повідомлень, яких достатньо аби задовольнити потреби розроблювано системи.

Було розглянуто основні принципи та терміни роботи Docker. Ця система є сучасним і зручним рішенням для розробки платформо незалежних систем. Також докер є дуже корисним при створенні розподілених систем, наприклад мікросервісів. Адже він надає функціонал для автоматизації та керування багатьох етапів розробки та роботи розподілених систем.

Було показано основу архітектури та головні складові, які забезпечують роботу: Docker командна утиліта, Docker Daemon та Docker Registry. Також були наведені найпопулярніші команди для роботи з ними.

Було розглянуто такі важливі поняття як образ та контейнер, відносини між ними та способи їх використання. Також було розглянуто томи як засіб для зберігання постійних файлів незалежно від стану контейнеру. Було показано важливий елемент для побудови розподілених систем Docker Network, яка дозволяє по різному будувати мережі між контейнерами.

Були показані переваги Docker Compose як технологія для спрощення процесу при роботі з багатьма сервісами.

Усі вище наведені особливості стануть у нагоді при створенні мікросервісних систем.

РОЗДІЛ 3
МОДЕЛЮВАННЯ СИСТЕМИ МОНІТОРИНГУ

3.1 Огляд призначення розроблюваної системи

В даній роботі буде розроблена мікросервісна система моніторингу фільмів. Дана система повинна надавати користувацький інтерфейс для перегляду фільмів в базі даних, REST API кінцеві точки та мати служби для моніторингу фільмів.

Моніторинг фільмів буде відбуватися за рахунок періодичного опитування стороннього ресурсу, який містить фільми та викачкою їх в базу даних застосунку у випадку, якщо є нові. Періодичність можна налаштовувати.

Користувачам буде надана можливість реєстрації та подальшої авторизації. Авторизовані користувачі можуть оцінювати фільми та залишати коментарі. Також вони можуть обирати цікаві для них жанри, та отримувати повідомлення про вихід нових фільмів з жанрами, на які підписані користувачі.

Ця система складатиметься з наступних частин та зв’язків між ними, що зображені на рисунку 3.1: API, що надає функціонал для перегляду фільмів, сервіси для викачки нових фільмів та розсилки повідомлень про це та клієнтської частини, яка буде використовувати API.

Як вимоги до основного функціоналу клієнта можна виділити наступні пункти:

1) перегляд списку фільмів;
2) перегляд детальної інформації про фільм;
3) пошук фільмів за ключовими словами;
4) сортування фільмів за певними критеріями;
5) форма реєстрації користувачів;
6) форма логіну;
7) підписка користувачів на жанри;

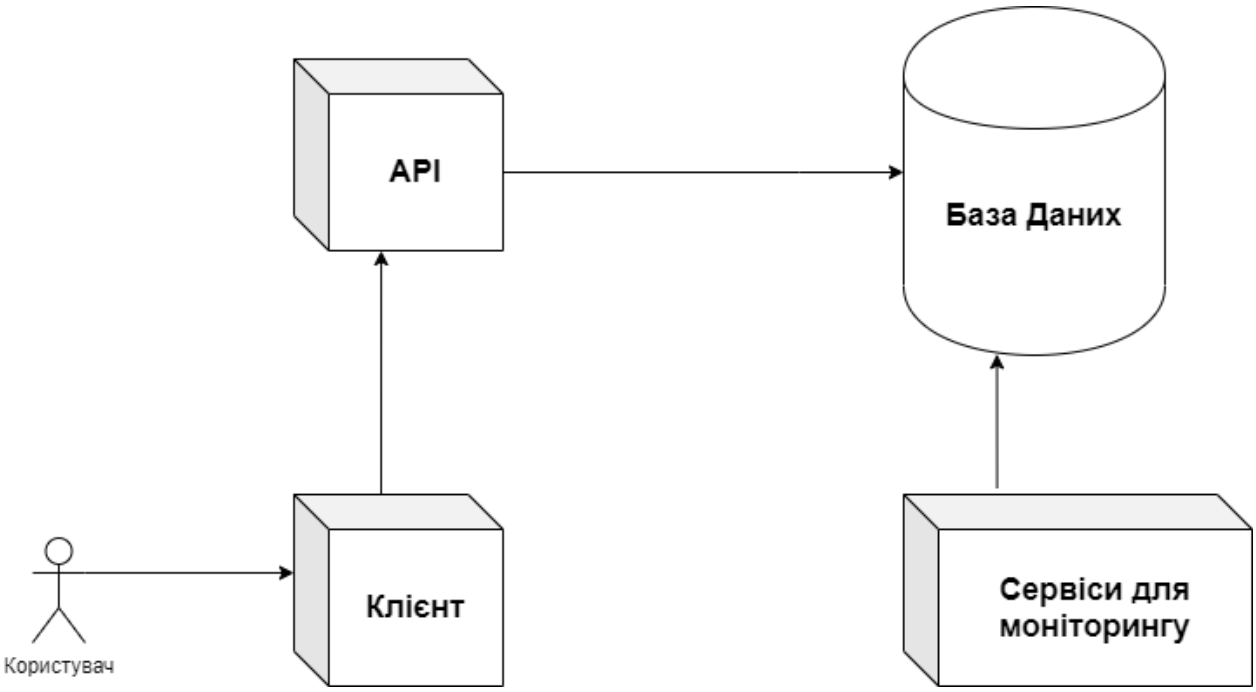


Рис 3.1. Взаємодія елементів верхнього рівня системи

3.2 Огляд технологічної бази

Мікросервісна система моніторингу фільмів є доволі складним програмним рішенням, тому вимагає сучасних технологій, які надають відповідний функціонал. Розглянемо технології які будуть використані в процесі розробки.

3.2.1. .Net Core як основна платформа для серверної розробки

.Net Core – це сучасна версія звичайної .Net платформи, що була створена Microsoft як рішення для незалежної від платформи розробки.

Серед переваг .Net Core можна виділити наступні пункти:

1) .Net Core є крос-платформною, на відміну від звичайного .Net. що означає, що її можна запускати на різних операційних системах, таких як Windows, MacOS, Linux;
2) велика кількість вбудованих та сторонніх готових до використання бібліотек;

3) наявність великої кількості синтаксичних конструкцій, що полегшують написання коду;
4) дана платформа активно підтримується та розвивається компанією Microsoft та спільнотою;
5) підтримка багатьох мов (C#, F#, Visual Basic), які компілюються в однаковий проміжний код, що дозволяє використовувати бібліотеки написані на одній мові в іншій.

Як мова програмування буде використаний C# - найпопулярніша в наш час мова для .Net. Це об'єктно-орієнтована мова, яка дозволяє швидку і зручно писати рішення поставлених задач. Також на відміну від багатьох подібних мов тут наявні синтаксично полегшені конструкції, які навіть дозволяють працювати з об'єктами, як в динамічних мовах.

Для розробки проекту в цій роботі використовується версія .Net Core 2.2, так як це найновіша версія на момент початку роботи.

3.2.2. Microsoft SQL Server як сервер бази даних

В розроблюваній системі буде існувати багато зв'язків, що є передумовою для використання реляційної бази даних.

Microsoft SQL Server – це система керування реляційними базами даних, яка розроблена компанією Microsoft. Як мова написання запитів використовується T-SQL (Transact-SQL), діалект, що є надбудовою над звичним SQL.

MS SQL Server має наступні переваги:

1) масштабованість – система може працювати як на невеликих портативних комп'ютерах так і в кластері;
2) максимальний розмір сторінок до 8 КБ, що забезпечує швидку обробку даних та зручну роботу зі складною структурою інформації;
3) можливість відміни запиту;

4) передова система безпеки Microsoft Baseline Security Analyzer;
5) наявність сервіс для індексного пошуку Full Text Search;
6) наявність інструментів для аналізу роботу системи та даних;
7) наявність системи звітів;
8) підтримка і зручна робота з такими продуктами Microsoft як Excel та Access;
9) дуже зручна система при розробці в середовищі .Net.

3.2.3. Angular як фреймворк для створення користувацького інтерфейсу

Розроблювана система буде мати користувацький інтерфейс. Для його написання оберемо фреймворк Angular.

Мова, що використовується в цьому фреймворку – TypeScript. Ця мова є об'єктно-орієнтованою строго типізованою надбудовою над JavaScript. TypeScript є гарним рішенням для великих проектів зі складно архітектурою, де необхідно підтримувати певні контракти. На відміну від JavaScript, він вказує на помилки при збиранні, тим самим надаючи захист від логічних помилок під час виконання, які набагато складніше нейтралізувати. Також ця мова надає зручний синтаксис, який має деякі схожі аспекти з Java, для написання об'єктно-орієнтованих рішень.

До переваг фреймворку Angular можна віднести наступні:

1) мова TypeScript;
2) зручна система створення та організації компонентів;
3) можливість змінювати існуючі теги;
4) розподіл логіки(скриптів), стилів та розмітки по різних файлах;
5) наявність cli інструментів, що дозволяє зручно створювати різні компоненти;
6) використання RxJS, що дозволяє зручно працювати з потоком даних в асинхронному режимі;

7) наявність Dependency Injection (укр. ін'єкція залежностей), що дозволяє створювати слабко зв'язані компоненти та зручно надавати їм необхідні залежності;
8) постійна підтримка;
9) велика кількість готових до використання компонентів та сервісів;

3.2.4. RabbitMQ як засіб для організації обміну повідомленнями між компонентами системи.

Для організації передачі повідомлень між сервісами буде використаний брокер RabbitMQ. Його особливості та переваги описані в розділі 2.2.5.2.1. Так як дана система буде мати декілька сервісів, то RabbitMQ ідеально підходить як рішення з багатьма готовими до використання інструментами. До того ж для RabbitMQ написано багато бібліотек під різні мови програмування та платформи, включно з .Net.

3.2.5. Docker як засіб автоматизації побудови та розгортки

Розроблювана система буде мати декілька концептуально різних частин, що майже повністю відрізняються у вимогах до середовища виконання. Тому Docker, який описаний у розділі 3, є необхідністю в такому випадку. Його інструменти значно пришвидшать процес розробки і спростять подальшу розгортку на сервері. Також буде використовуватися механізм Docker Compose для ще більш зручної роботи з багатьма частинами.

3.2.6. TMDB API

В даній системі фільми будуть викачуватися з стороннього ресурсу, за допомогою періодичного його опитування. Цим ресурсом виступе TMDB, який надає зручне API для інтеграції у застосунки.

TMDB (The Movie Database) API – спеціальне API сайту The Movie Database, яке надає широкий функціонал для роботи з фільмами. Для того

аби отримати доступ до API необхідно зареєструватися на сайті та отримати API ключ, який необхідно передавати при кожному запиті на ресурс.

Серед основних кінцевих точок TMDB, можна виділити наступні:

1) /discover – пошук за певними параметрами;
2) /find – отримання за певним id;
3) /search – пошук за ключовими словами.

Для кожного з цих методів можна дописати наступний сегмент URL, наприклад movie(фільм), genre(жанр), actor(актор) та інші, який буде визначати до яких об’єктів буде запит.

3.3 Огляд архітектури системи

Архітектурою розроблюваної системи є мікросервіси навколо моноліту. В даному випадку монолітом буде виступати WebApi частина системи. А мікросервісами будуть сервіси призначені для викачки фільмів та розсилки відповідних повідомлень. В додатку А.3 зображено архітектуру системи та взаємозв’язки між основними частинами.

Центральним компонентом виступає RabbitMQ брокер, який призначений для організації взаємодії між компонентами і при цьому збереження їх незалежності один від одного. Також зображено черги (queues), ключи прив’язки (binding key) та точки обміну (exchanges) і те як частини системи взаємодіють з ними.

В додатку А.3 зображені наступні компоненти:

1) Scheduler;
2) MailSender;
3) Notifier;
4) DataLoader;
5) UserSubscriptionsService;
6) WebApi.
7) Client.

Як було сказано вище WebApi слугує для надання функціоналу користувацького доступу до існуючих фільмів. Це API використовується напряму клієнтською частиною.

Scheduler виступає як сервіс, що відповідальний за запуск певних процесів в системі. В даній системі існує один такий процес, який відбувається в сервісу DataLoader. Але за необхідності можна додавати нові.

DataLoader є головним компонентом і містить в собі необхідний функціонал для опитування TMDB API та занесення фільмів в базу даних.

UserSubscriptionsService слугує для обробки підписок користувачів. Він отримує фільми, що вийшли і переглядає користувачів, що підписані на жанр фільму, таким чином формує список користувачів, яким необхідно відправити повідомлення.

MailSender розсилає повідомлення на пошту відповідним користувачам. Цей сервіс працює з повідомленнями в загальному вигляді, тобто він не несе логіки по створенню, а лише відсилає їх. Тож він може відправляти не лише повідомлення про нові фільми, а й будь-які інші.

Notifier відсилає повідомлення на клієнтську частину в режимі реального часу за допомогою WebSocket про вихід нових фільмів незалежно від користувача та його підписок.

Client – клієнт для зображення даних в браузері та надання функціоналу користувачам для взаємодії з системою.

Для полегшення імплементації мікросервіси використовують одну базу даних. На рисунку 3.2 зображено зв'язок сервісів із базами даних.

В додатку А.2 зображено діаграми баз даних SchedulerJobDB (таблиці Trigger та Job) та MovieDB (всі інші таблиці).

SchedulerJobDB – база даних для зберігання процесів, які слід запустити Scheduler сервісу та тригерів часу запуску.

MovieDB – база даних для зберігання основних даних системи: фільмів, акторів, користувачів та інше.

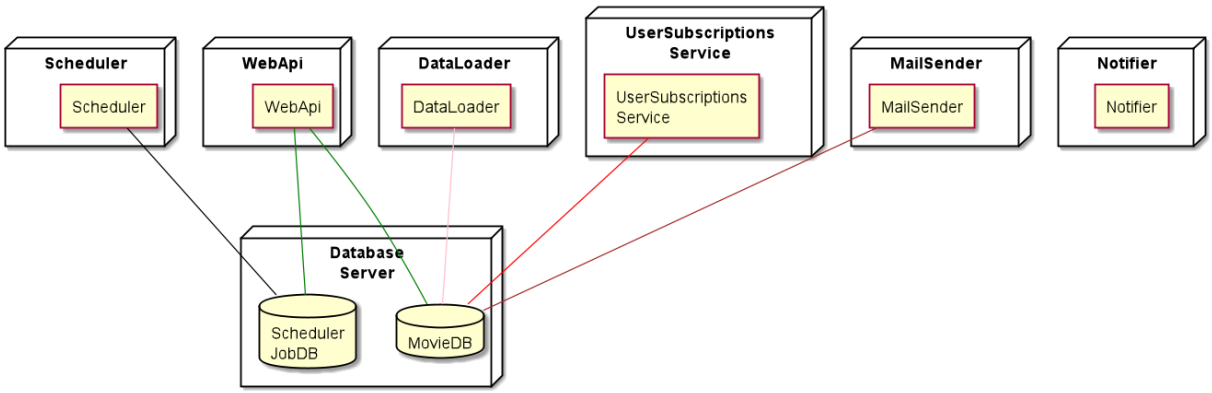


Рис 3.2. Зв'язок вершин системи з базами даних

3.4 Бізнес-кейси для оновлення бази фільмів

Одним з головних процесів в системі є оновлення фільмів. Адже в нього входить зміна часу оновлення, власне оновлення, генерація листів користувачам, що підписані на жанри. Загальний алгоритм цього процесу зображено в додатку А.1. Цей процес можна розбити на наступні етапи.

1. Адміністратор ініціює зміну часу через API за допомогою кінцевої точки /jobs/{id процесу}/trigger, як зображено на рисунку 3.3. Як дані в цьому запиті передається крон. Користувач повинен бути авторизованим та мати необхідні права, тому ще слід передати в хедері Authorization JWT токен, отриманий як результат авторизації. Це POST запит і JSON тіло може виглядати наступним чином: {"cron": "30 05 7 * * ?" }

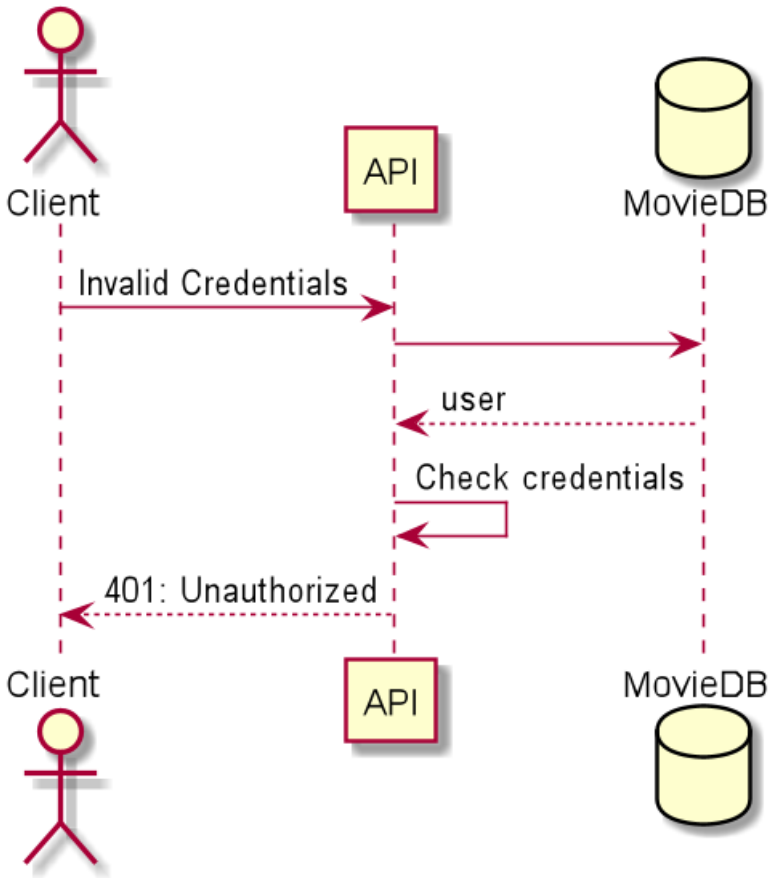


Рис 3.3. Бізнес кейс зміни крону

2. Scheduler сервіс отримує повідомлення з новим кроном. Він оновлює час виконання закачки фільмів в базі даних та за його настання відправляє повідомлення про це до брокера, як зображено на рисунку 3.4.

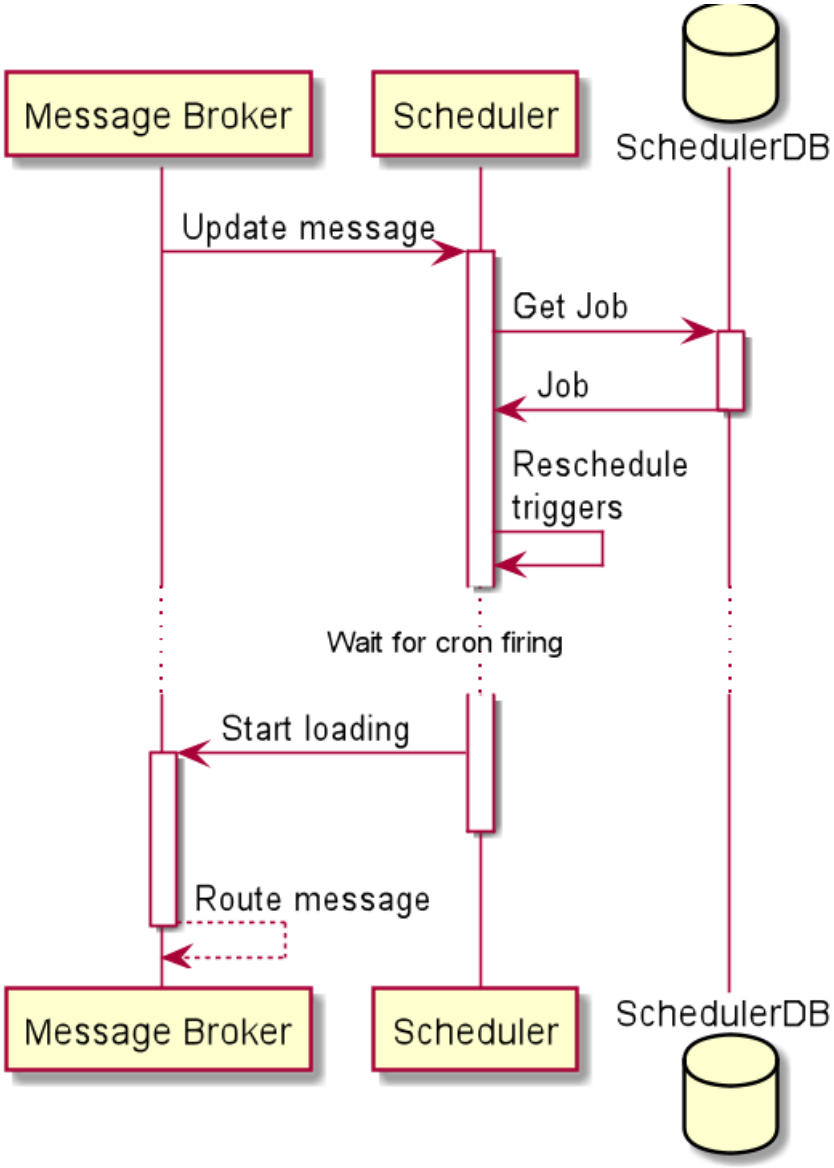


Рис 3.4. Бізнес кейс зміни тригеру та надсилання повідомлення про запуск

3. DataLoader сервіс, отримавши повідомлення про початок, починає процес завантаження, що зображено на рисунку 3.5. Він починає опитування TMDB API посторінково. Перевіряє фільми з кожної сторінки на наявність в базі даних застосунку. Якщо серед отриманих є ті, яких немає в базі, то вони додаються. Цей процес закінчиться, коли на сторінці не буде нових фільмів. Потім повідомлення зі списком нових фільмів відправляється в брокер.

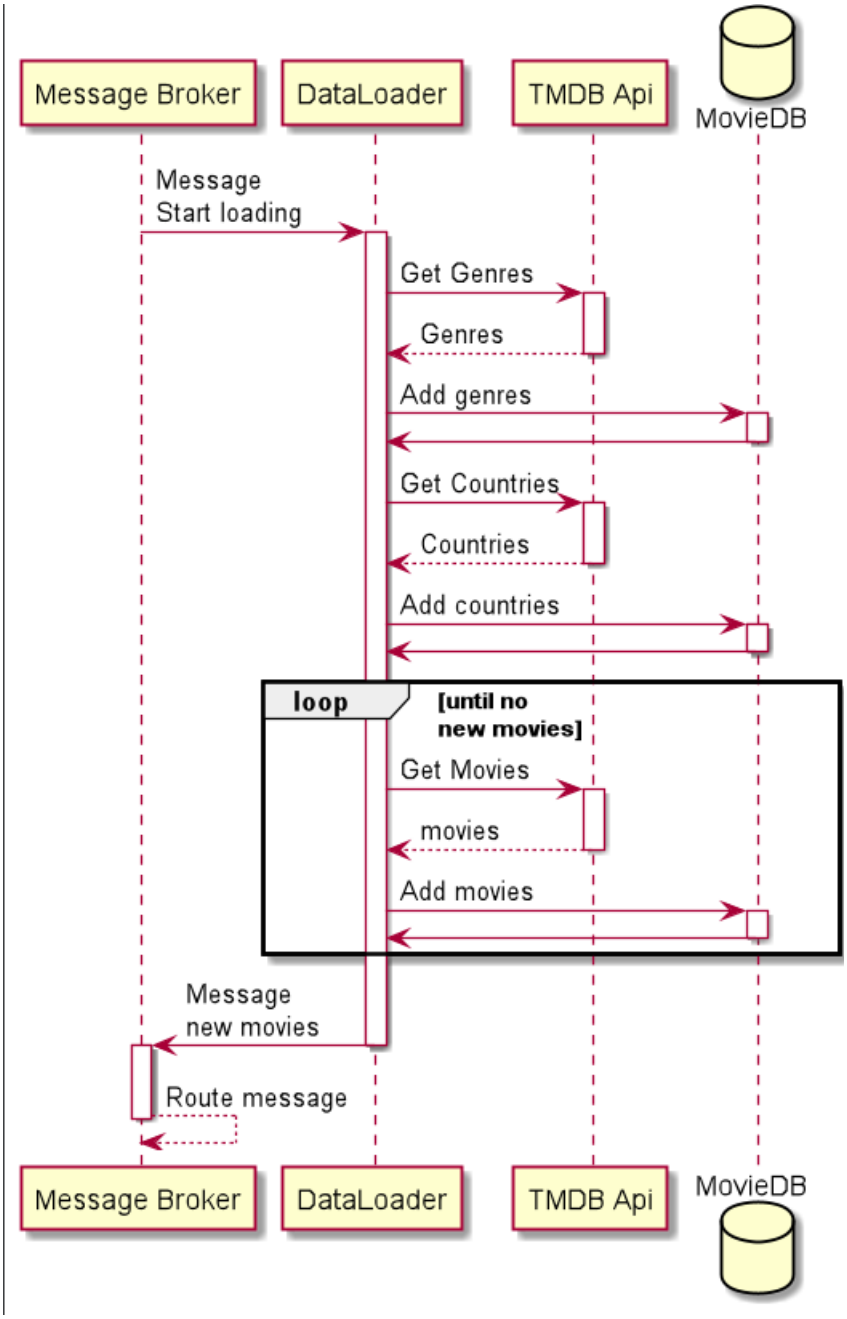


Рис 3.5. Бізнес кейс завантаження фільмів

4. UserSubscriptionsService отримує повідомлення зі списком фільмів, як зображено на рисунку 3.6. Потім опрацьовує ці фільми і користувачів, які підписані на відповідні жанри. З опрацьованих даних сервіс формує повідомлення, які будуть надіслані користувачам. Потім він надсилає повідомлення про необхідність розсилки повідомлень.

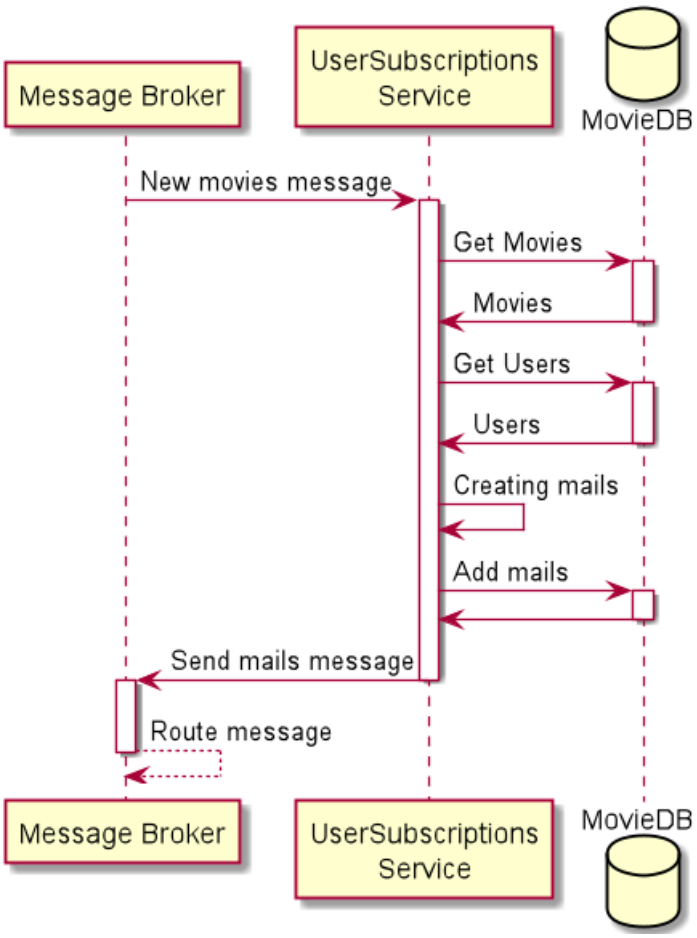


Рис 3.6. Бізнес кейс обробки підписок користувачів

5. Notifier сервіс отримує повідомлення від брокера з фільмами, що вийшли. Підраховує їх кількість та надсилає за допомогою встановленого з клієнтом WebSocket зв'язку повідомлення з кількістю фільмів. Для організації сокету використовується популярна бібліотека SignalR як на клієнті так і на сервері. Цей процес зображено на рисунку 3.7.

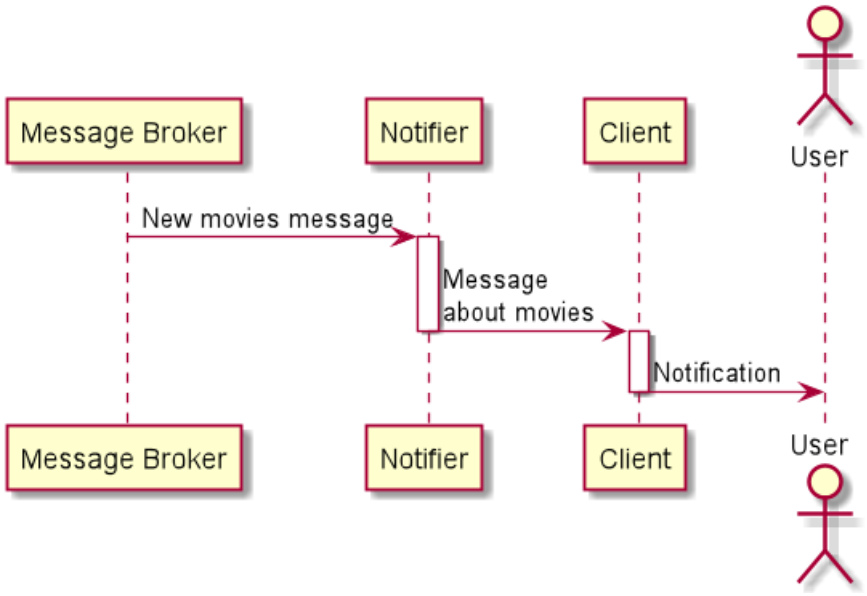


Рис 3.7. Бізнес кейс повідомлення клієнта про нові фільми

6. MailSender сервіс отримує повідомлення від брокера про необхідність надсилання електронних листів користувачам. Сервіс бере з бази усі невідправлені повідомлення та надсилає їх за допомогою SMTP протоколу, як зображено на рисунку 3.8.

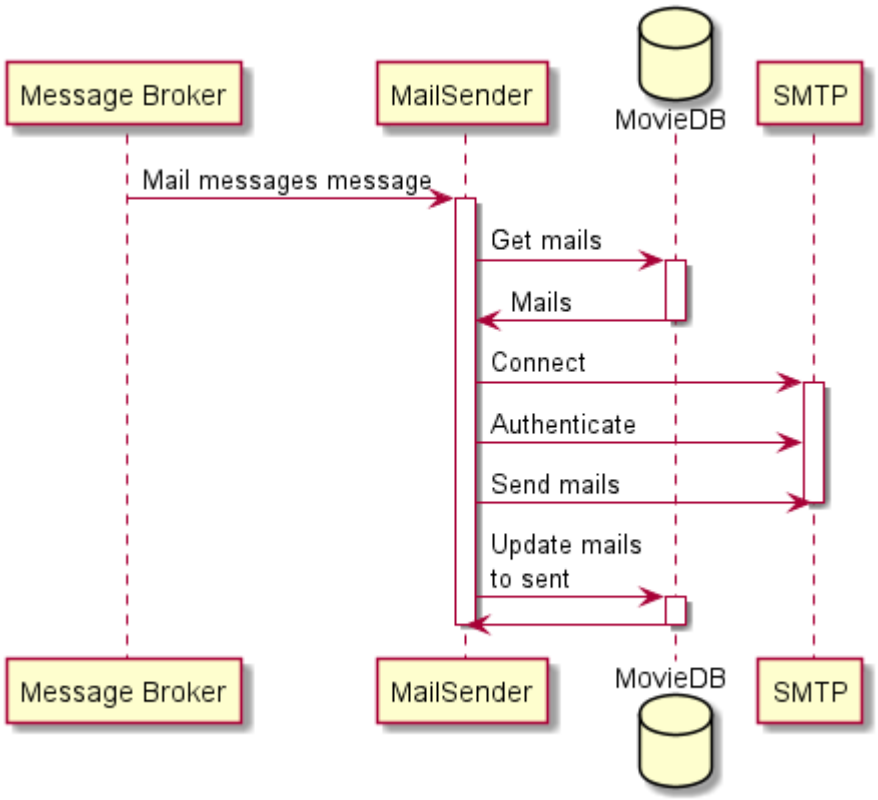


Рис 3.8. Бізнес кейс відправки листів на пошту

Як результат виконання всіх цих операцій користувач, що підписаний на певні жанри отримує повідомлення на пошту (рисунок 3.9) і також клієнтський додаток показує повідомлення(рисунок 3.10) з кількістю нових фільмів всім користувачам , включно з неавторизованими.

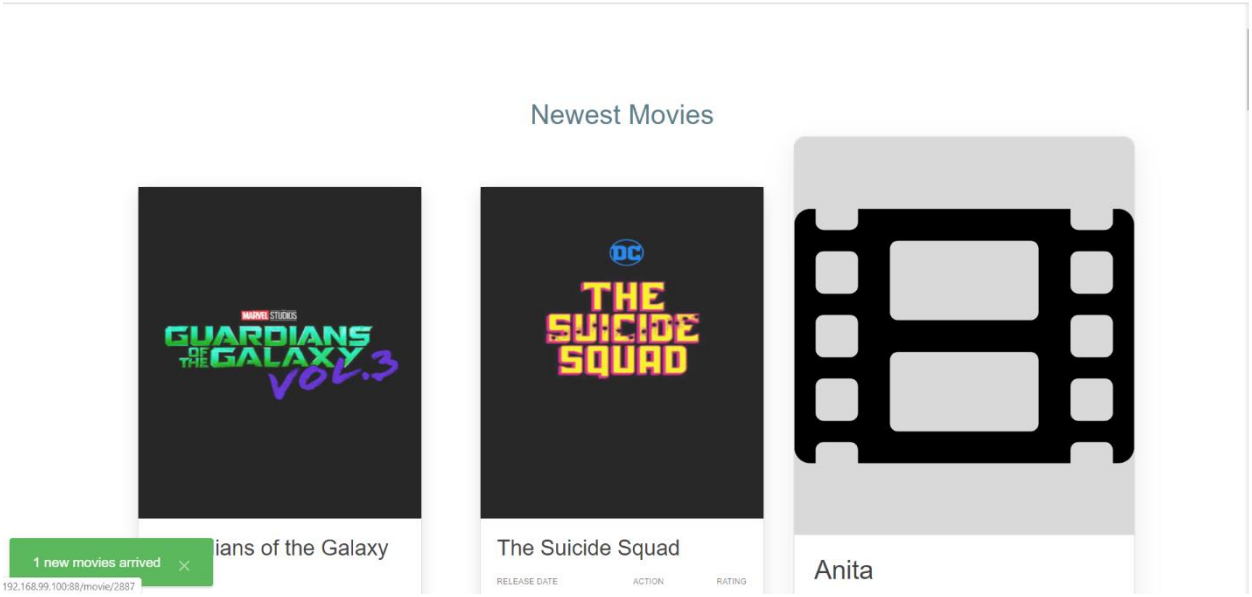


Рис. 3.9. Нотифікація на клієнтській частині

Рис. 3.10. Лист на пошті про вихід нового фільму

Висновки до розділу 3

Було розроблено систему з використанням мікросервісної архітектури та використання технологій, що описані в попередніх розділах а саме RabbitMQ та Docker. Для реалізації були використані платформа .Net Core з мовою C#, система керування реляційними базами даних Microsoft SQL Server, фреймворк для розробки клієнтської частини Angular. Архітектурна організація системи – це мікросервіси навколо моноліту.

Основною функцією системи є періодичне опитування та викачка фільмів зі стороннього ресурсу TMDB API, для цього призначений набір мікросервіс, які передають повідомлення між собою за допомогою брокера RabbitMQ, який виступає посередником.

В системі присутнє REST Web API, яке реалізоване як моноліт, і містить в собі різноманітний функціонал, такий як реєстрація, авторизація, перегляд фільмів, сортування, пошук, підписка на жанри, рейтинги, коментарі та інші. Це API споживає клієнт, написаний на Angular, який реалізує вище перераховані функції в графічному вигляді, зручному для користувачів.

Реалізація частини системи у вигляді мікросервісів дала важливі переваги: незалежність розробки та розвитку окремих частин, підтримка робочого стану системи при проблемах з певною її частиною, гарна масштабованість, зручне розширення (додання нових сервісів) та розгортка в хмарних сервісах.

Для побудови та розгортки системи було використано Docker та інструмент docker compose, за допомогою яких ці процеси значно полегшилися та пришвидшилися. В майбутньому за по треби більш тонкого керування ресурсами, що виділяються під кожну вершину, та кількістю сервісів можна використовувати більш допрацьовані інструменти, такі як Kubernetes.

РОЗДІЛ 4
ОГЛЯД ТА ТЕСТУВАННЯ РОБОТИ СИСТЕМИ

Розроблювана система має різноманітний функціонал, тому тестування є необхідною частиною розробки. Більшість тестів було зроблено за допомогою unit тестів та ручного тестування.

4.1 Unit тестування

В ході розробки було написано 169 unit тести для сервісної частини. Плюсом таких тестів є відносна простота написання та сконцентрованість логіки на найменших частинах (класах в даному випадках). Для написання тестів було використано xUnit бібліотеку. Для того аби імітувати інтеграцію з іншими частинами було використано бібліотеку Moq. Результат успішного проходження тестів по кожному з проектів зображено на рисунку 4.1.

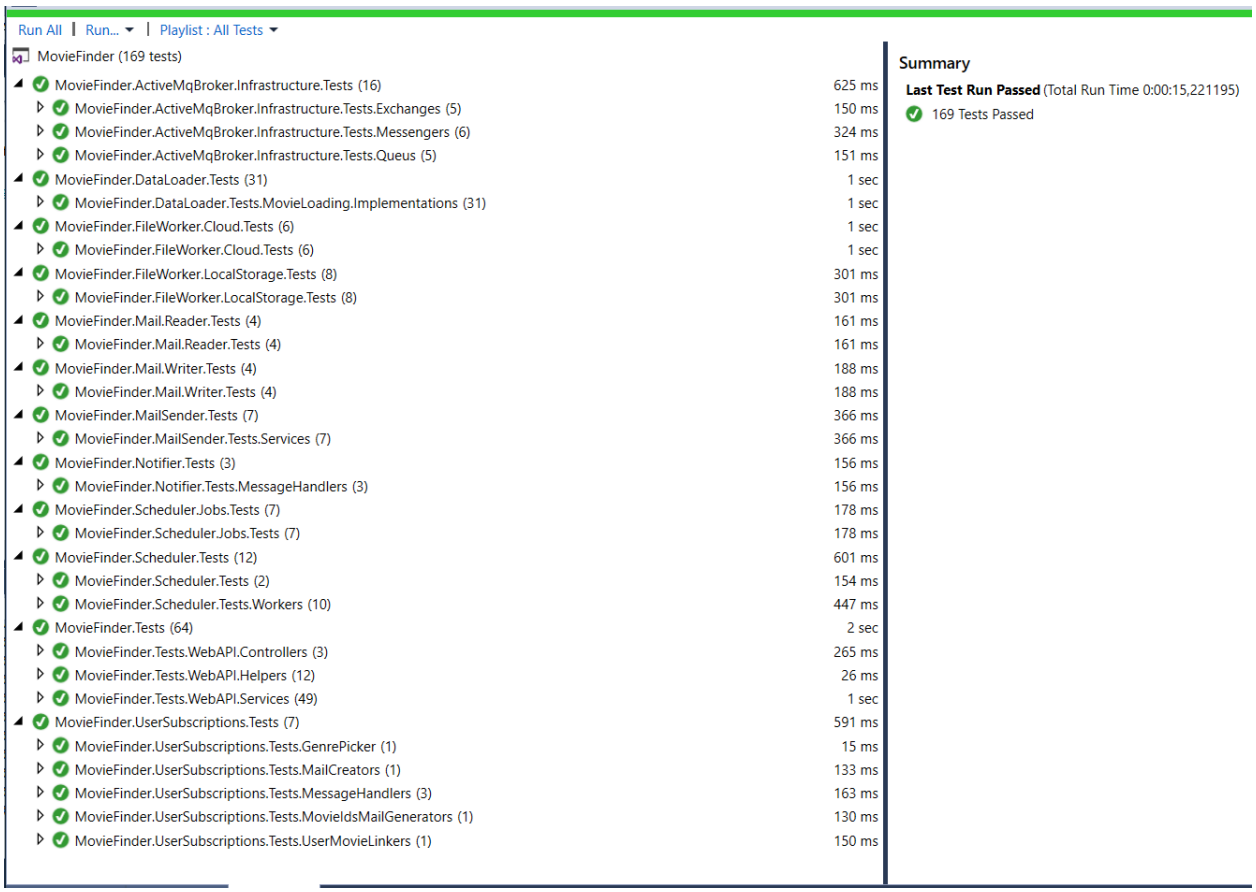


Рисунок 4.1 Результат виконання Unit тесів

4.2 Логування подій в системі

Для більшості етапів запуску та різних можливих подій написано логіку логування в консоль та у файли для того аби отримувати інформацію про те, що відбувається в системі або досліджувати природу виникнення помилки. Приклад логів, які можна побачити при успішному запуску системи за допомогою docker-compose зображено на рисунку 4.2. Приклад логування подій у сервісі Scheduler зображено на рисунку 4.3.

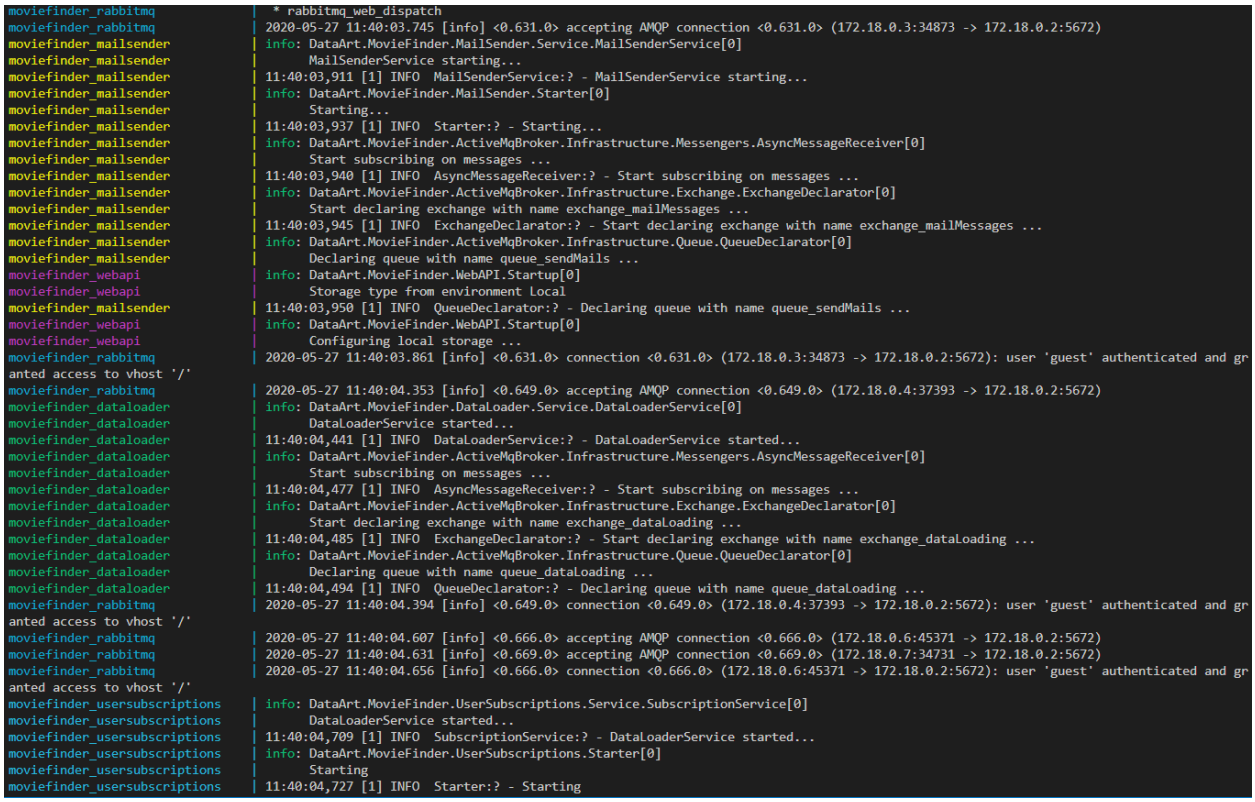


Рис. 4.2. Приклад логів при запуску системи

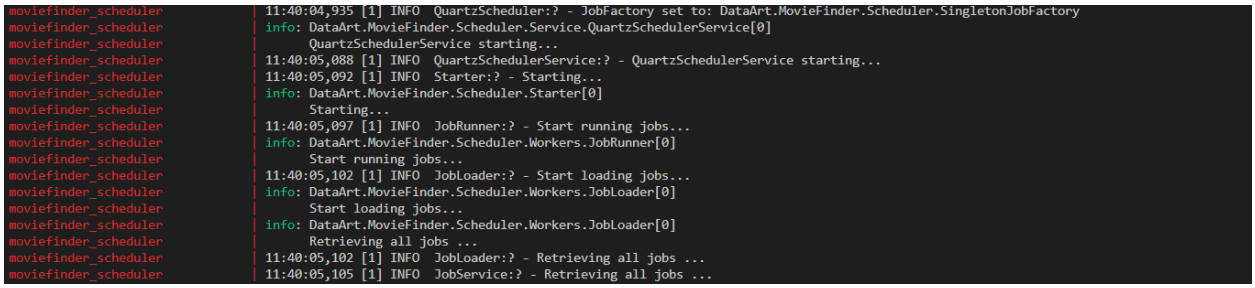


Рис. 4.3. Приклад логування подій у Scheduler сервісі

4.3 Графічне представлення

Для роботи з фільмами було розроблено графічний клієнт на Angular, який надає функціонал для використання кінцевих точок API. Оглянемо деякі з доступних функціоналів на клієнті.

4.3.1 Реєстрація та вхід в систему

Для того аби мати доступ до певних функцій системи користувач повинен зареєструватися за допомогою спеціальної форми, що зображена на рисунку 4.4. Користувач повинен правильно заповнити поля форми, які є небхідними. Підписка на жанри одна з основних функцій системи.

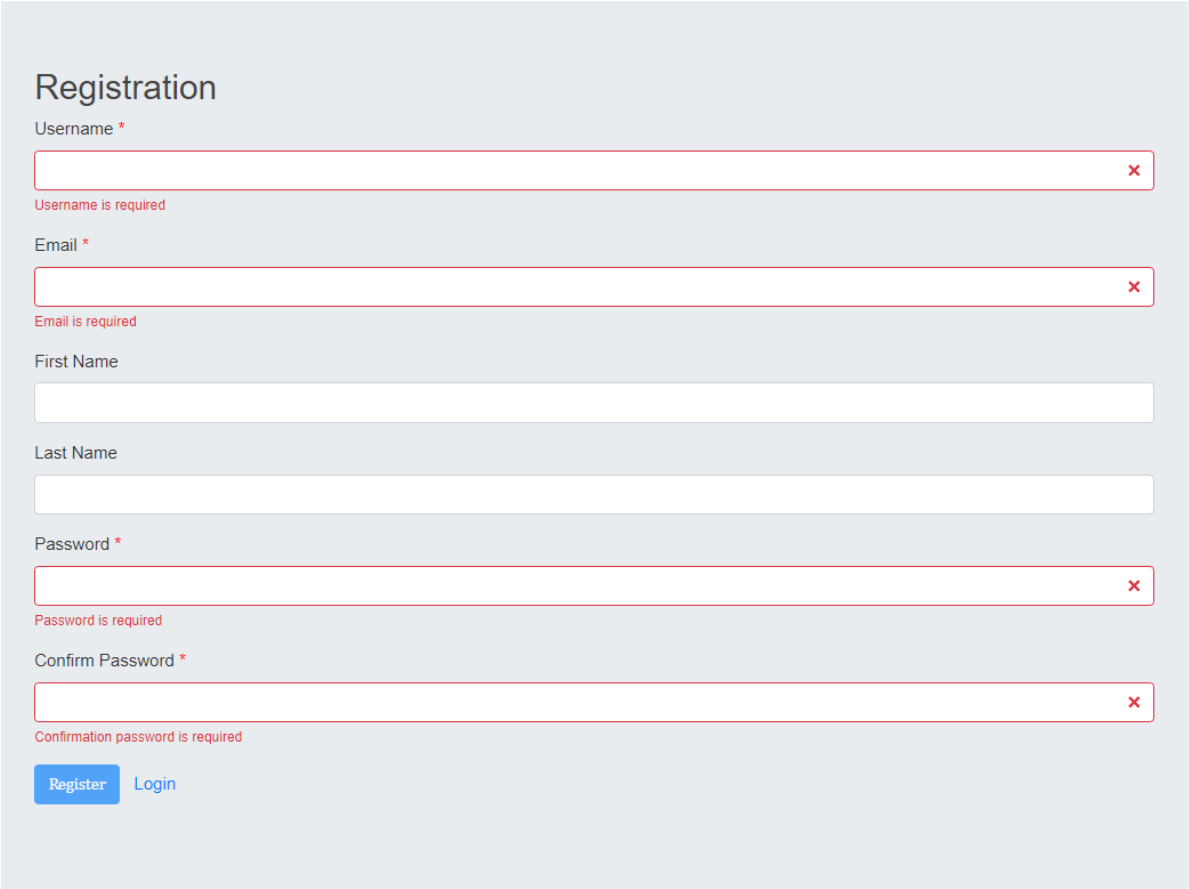


Рис 4.4. Форма реєстрації

Після реєстрації користувач отримує змогу увійти в систему, та йому відкривається доступ до таких функцій як написання коментаря та надання

рейтингу фільму. Процес входу виконується за допомогою форми входу, що зображена на рисунку 4.5.

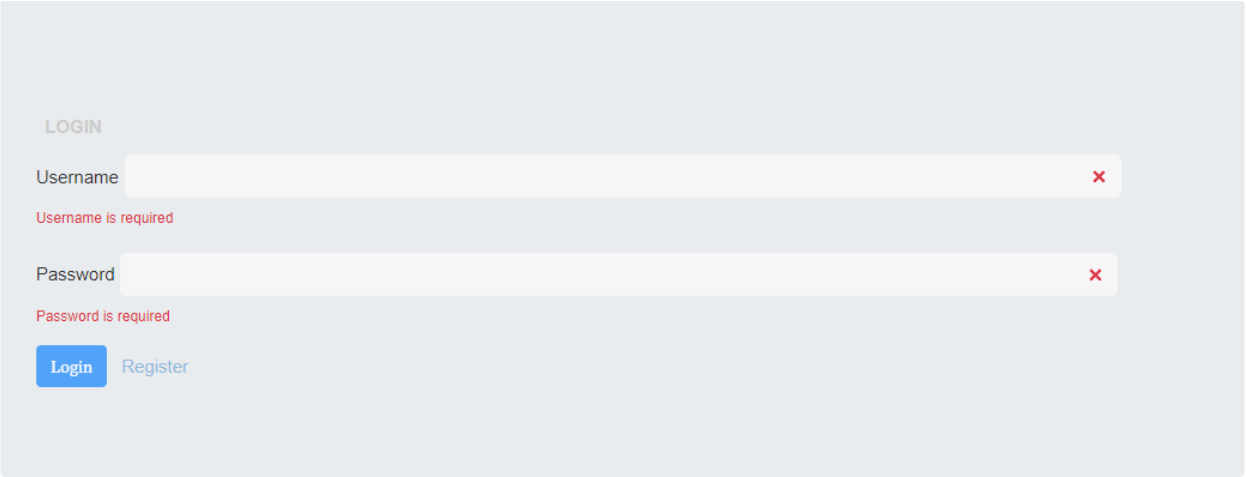


Рис. 4.5. Форма входу

4.3.2 Підписка на жанри

Користувачам, що увійшли до системи надається змога обирати жанри, що їх цікавлять, для того, аби згодом отримувати повідомлення про вихід нових фільмів за цими критеріями. Обирати жанри можна на сторінці користувача, як зображено на рисунку 4.6.

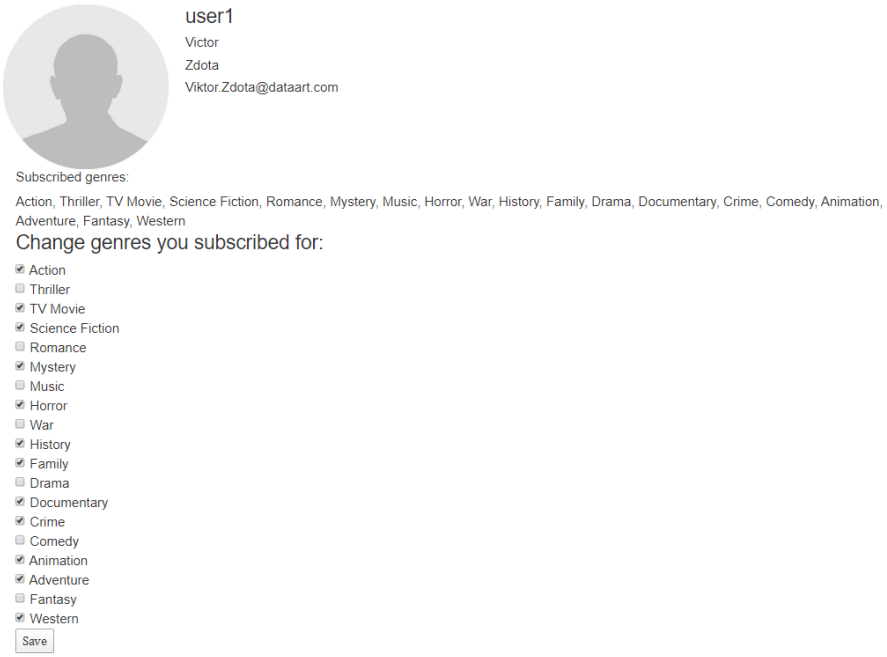


Рис. 4.6. Інтерфейс для підписки на жанри

4.3.3 Оцінка та коментарі до фільмів

Авторизовані користувачі мають змогу ставити оцінку фільму та залишати коментарі на сторінці фільму (рисунок 4.7).

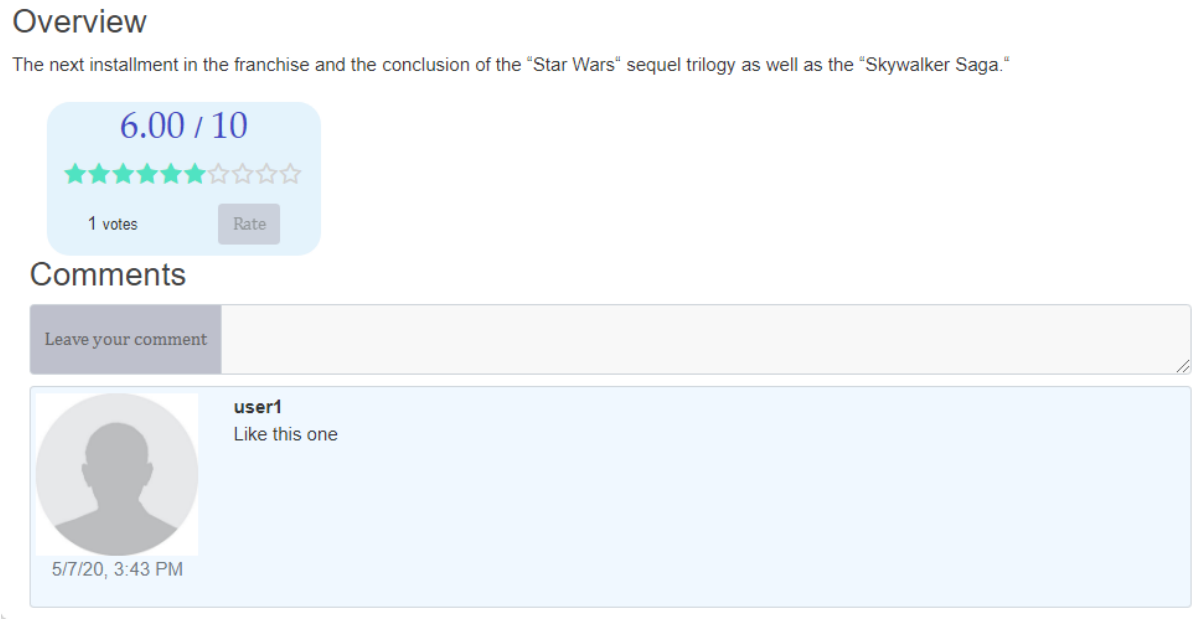


Рис. 4.7. Оцінка та коментарі до фільму

4.3.4 Пошук за ключовими словами

Фільми та акторів можна шукати за ключовими словами, для цього реалізований виклик Full Text Search функцій.

Пошук здійснюється як динамічно, тобто під час введення фрази (рисунок 4.8), так і при натисканні кнопки “Search” для показу сторінки з результатами (рисунок 4.9 та 4.10).

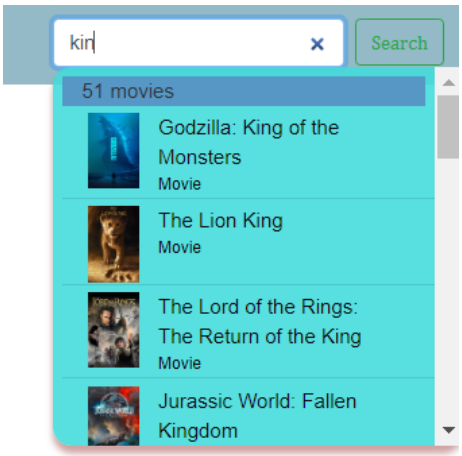


Рис. 4.8. Рядок пошуку з динамічними результатами

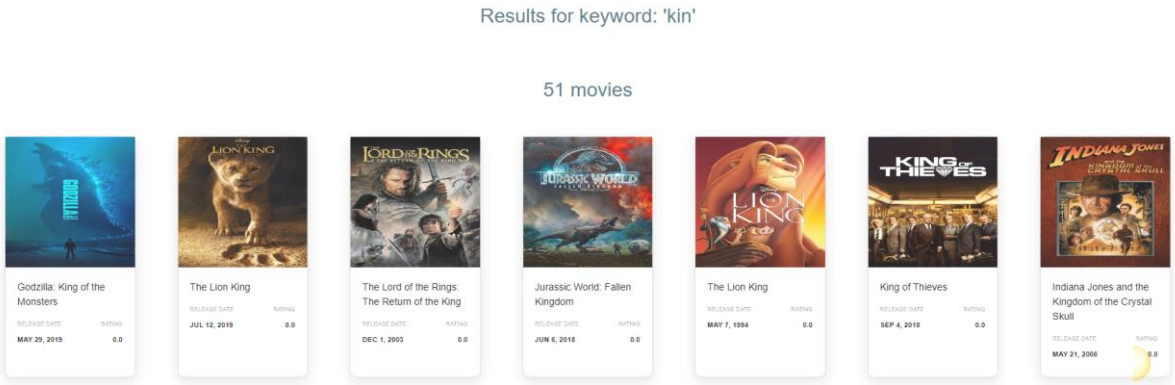


Рис. 4.9. Верхня частина результату пошуку за ключовим словом для фільмів

Рис. 4.10. Верхня частина результату пошуку за ключовим словом для акторів

Висновки до розділу 4

В даному розділі було оглянуто основні можливості та протестовано функціонал розроблюваної системи.

За допомогою unit тестів було покрито дрібні частини та досягнуто автоматизацію перевірки працездатності певних класів. За допомогою бібліотек для імітації роботи зв’язаних компонентів було протестовано адеватність взаємодії між частинами при різних варіантах роботи. Всі 169 тестів пройшли успішно, не виявивши помилок. Unit тести писалися в процесі розробки, завдяки чому, після деяких змін до коду, було дуже легко перевірити, чи ці зміни не викликали помлок.

Також при написанні коду в багатьох місцях було включено логіку для логування, завдяки чому можна спостерігати інформації про стан тієї чи іншої частини системи в консолі або у файлі. Логування виявилося дуже корисним при знайденні помилок та налогодження роботи. При аналізуванні результатів логування кінцевого варіанту системи був очікуваний результат.

Весь функціонал системи, що представлений на клієнтській частині було протестовано вручну. Всі аспекти клієнтської частини, такі як реєстрація, вхід до системи, перегляд фільмів, коментування та інше працюють справно.

ВИСНОВКИ

Дана робота присвячена побудові системи моніторингу фільмів, що базується на мікросервісній архітектурі, і також надає функціонал та графічний інтерфейс для перегляду інформації.

При розробці були використані наступні технології: RabbitMQ, Docker та власне мікросервісна архітектура.

Було наведено особливості систем, що базуються на мікросервісах та перераховано основні переваги такого підходу: гарна масштабованість, незалежність розробки окремих частин, зручне розширення, простота логіки окремого мікросервіса, економне використання комп'ютерних ресурсів при розподіленій структурі, стабільність роботи системи при виході за ладу деяких частин. Також були показані принципи, яким слідують такі системи: невеликий розмір, незалежність, обмеженість бізнес контексту, використання шаблону Smart Endpoints And Dump Pipes, використання підходу Design For Failure, децентралізація, використання систем автоматизації в процесах розробки та підтримки, ітераційний розвиток.

Як зручне рішення для організації передачі повідомлень між мікросервісами було використано популярний брокер RabbitMQ. Було порівняно його з іншою популярною альтернативою Apache Kafka, в ході чого було встановлено, що RabbitMQ краще підходить під потреби розроблюваної системи за рахунок наступних переваг: гнучка система маршрутизації повідомлень, гарна відмовостійкість, наявність багатьох різноманітних вбудованих інструментів. До того ж трафік повідомлень в системі не високий, тож можливостей RabbitMQ цілком достатньо.

Для організації побудови та розгортки було використано Docker. Були описані основні принципи, частини та інструменти, які надає ця система. Серед функціоналу Docker, який знадобився при реалізації розроблюваної системи можна виділити наступне: автоматизація побудови та розгортки контейнерів сервісів, незалежність від платформи запуску, гнучке

налаштовування середовища під кожний контейнер, наявність в публічному реєстрі готових образів під різноманітні випадки, автоматизація роботи з системами, що складаються з багатьох частин.

Система моніторингу фільмів була розроблена за архітектурою мікросервіси довкола моноліту, де монолітом виступало API для пошуку та перегляду існуючих фільмів в системі, а мікросервіси виконували роль моніторингу, адаптацію моделей, та викачки фільмів зі стороннього ресурсу в базу даних застосунку та як результат цього процесу генерування повідомлень та їх розсилка користувачам, що підписані на певні жанри.

При розробці переваги мікросервісів дали гарний приріст в швидкості додання нових функціональних можливостей, зручну розгортку та керування системою, незалежність окремих частин одна від одної. В подальшому за необхідності таку систему буде зручно розмістити в хмарних сервісах, на окремих віртуальних машинах, що дає значний приріст в економії у порівнянні з суцільним монолітом.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ:

1. Sam Newman. Building Microservices: Designing Fine-Grained Systems / Sam Newman. – O'Reilly Media, 2015.

2. Morgan Bruce, Paulo A. Pereira. Microservices in Action / Morgan Bruce, Paulo A. Pereira. – Manning Publications, 2018.

3. Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, Mike Amundsen. Microservice Architecture: Aligning Principles, Practices, and Culture 1st Edition / Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, Mike Amundsen, Matt Mclarty. – O'Reilly Media, Inc, 2016.

4. Saineshwar Bageri. Learning Rabbit MQ with C#: A magical tool for the IT world/ Saineshwar Bageri. – BPB Publications, 2018.

5. Dotan Nahum, Emrah Ayanoglu, Yusuf Aytas. Mastering RabbitMQ / Dotan Nahum, Emrah Ayanoglu, Yusuf Aytas – Packt Publishing, 2016.

6. Gavin M. Roy. RabbitMQ in Depth / Gavin M. Roy – Manning Publications, 2017.

7. Jeff Nickoloff, Stephen Kuenzli. Docker in Action / Dotan Nahum, Emrah Ayanoglu, Yusuf Aytas – Manning Publications, 2016.

8. Sébastien Goasguen. Docker Cookbook: Solutions and Examples for Building Distributed Applications / Sébastien Goasguen – O'Reilly Media, Inc, 2016.

9. Dustin Metzgar. .NET Core in Action / Dustin Metzgar, Scott Hanselman – Manning Publications, 2018.

10. Andrew Troelsen, Philip Japikse. Pro C# 7 With .NET and .NET Core / Andrew Troelsen, Philip Japikse – Appress, 2017.

11. Andrew Lock. ASP.NET Core in Action / Andrew Lock – Manning Publications, 2018.

12. Adam Freeman. Pro Angular 6 / Adam Freeman – Appress, 2018.

13. Itzik Ben-Gan. T-SQL Fundamentals, Third Edition / Itzik Ben-Gan – Microsoft Press, 2016.

14. Microservice Architecture. [Електронний ресурс]. – Режим доступу: URL: https://microservices.io.

15. Microservices Solution. [Електронний ресурс]. – Режим доступу: URL: https://smartbear.com/solutions/microservices/.

16. RabbitMQ Documentation and Guides. [Електронний ресурс]. – Режим доступу: URL: https://www.rabbitmq.com/.

17. TMDB API Documentation [Електронний ресурс]. – Режим доступу: URL: https://www.themoviedb.org/documentation/api.

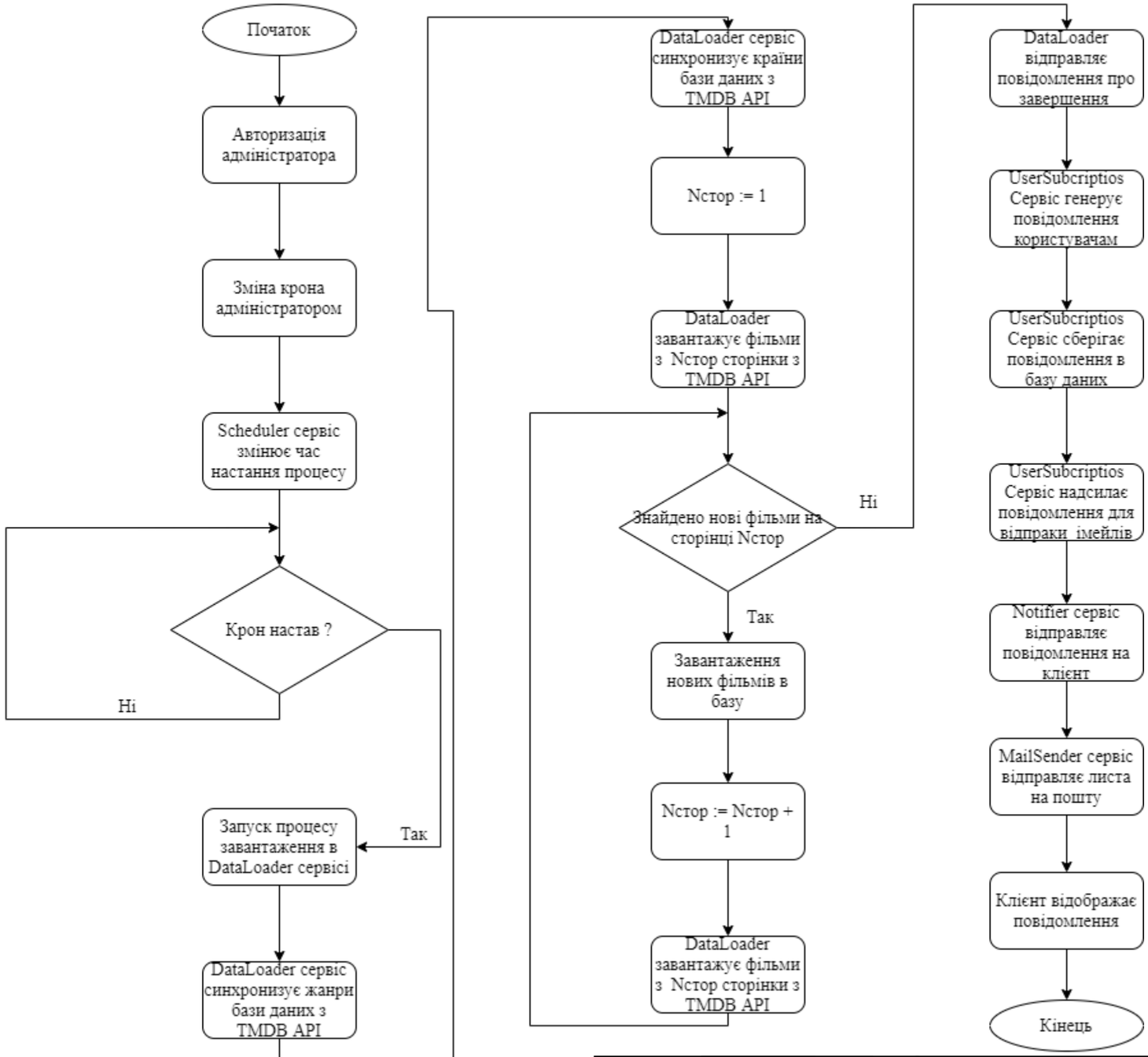
18. Docker Documentation [Електронний ресурс]. – Режим доступу: URL: https://docs.docker.com/.

19. .Net Core Documentation [Електронний ресурс]. – Режим доступу: URL: https://docs.microsoft.com/en-us/dotnet/core/.

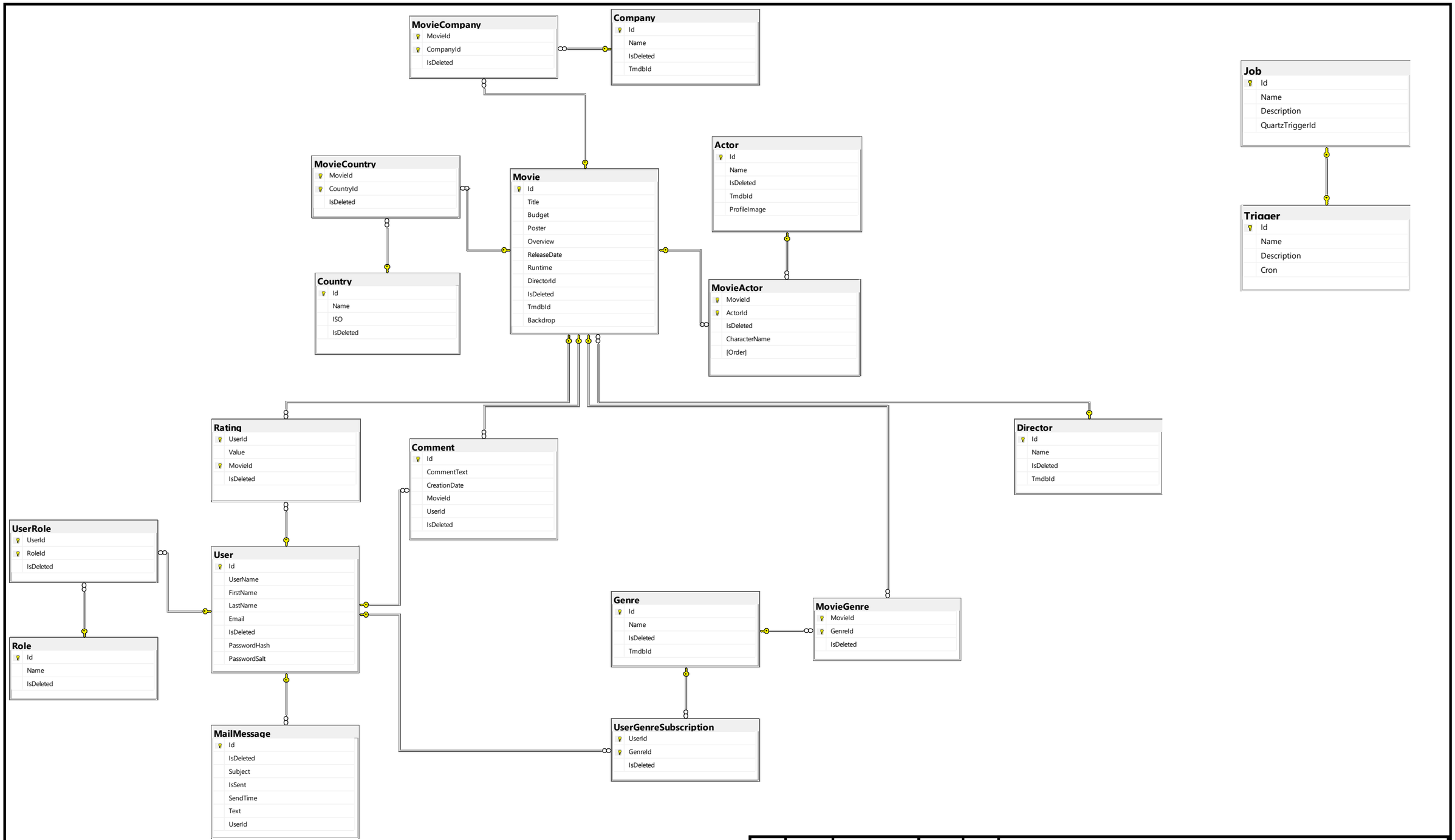
20. Angular Documentation and guides [Електронний ресурс]. – Режим доступу: URL: https://angular.io/docs.

20. MS SQL Full Text Search Documentation [Електронний ресурс]. – Режим доступу: URL: https://docs.microsoft.com/en-us/sql/relational-databases/search/full-text-search?view=sql-server-ver15.

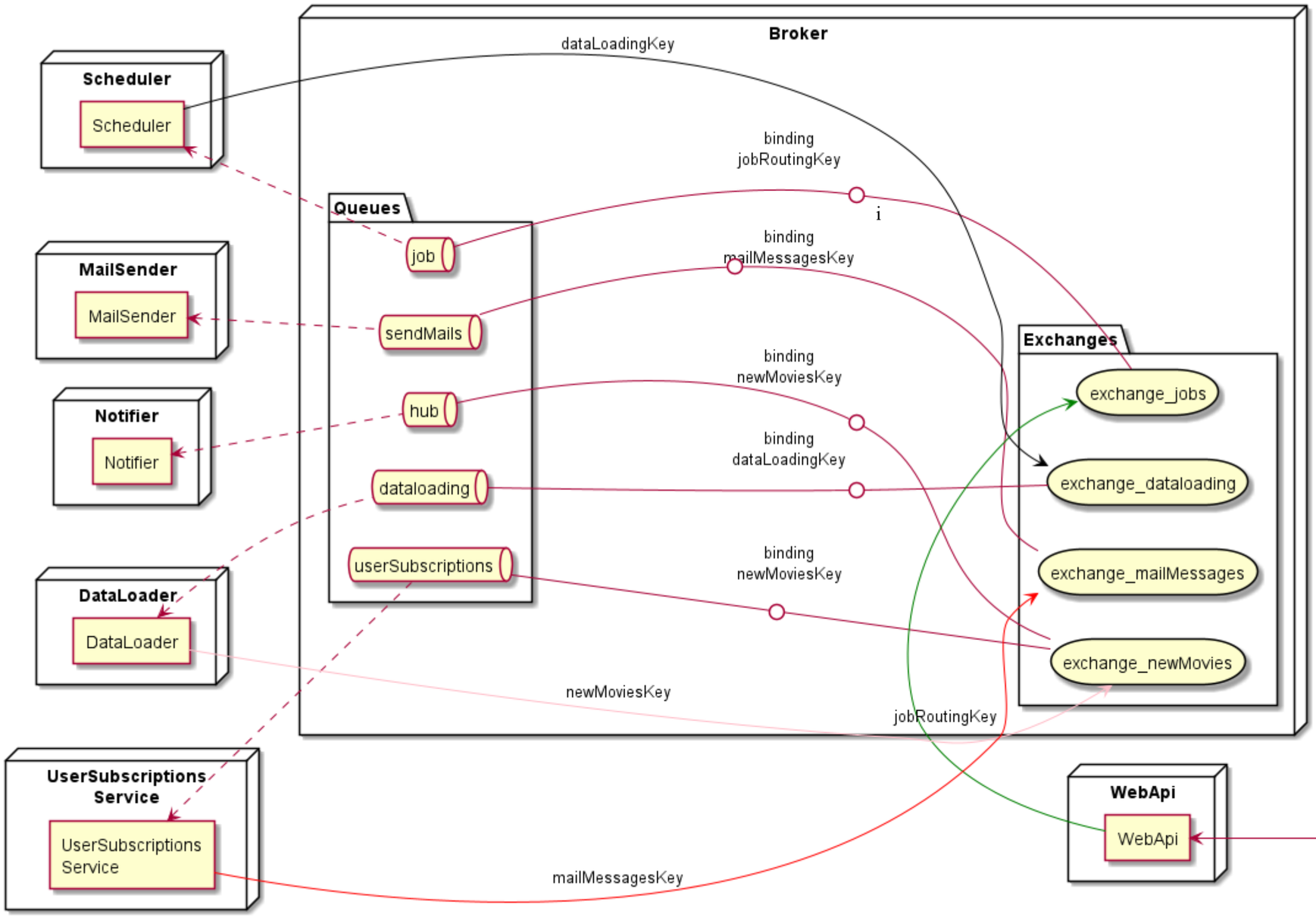
ДОДАТОК А



					ІАЛЦ.006310.004 Д1	
					Алгоритм оновлення фільмів Схема принципова	Літера / Маса / Масштаб
Зм.	Арк.	№ докум.	Підпис	Дата		
Розроб.		Здота В.О.				Аркуш 1 / Аркушів 1
Перевір.		Калюжний О. О.				НТУУ "КПІ" ФІОТ Група ІО-63
Т. контр.						
Н. контр.		Сімоненко В. П.				
Затв.						



					ІАЛЦ.006310.005 Д2
					Схема зв’язків у базах даних Схема функціонльна
Зм.	Арк.	№ докум.	Підпис	Дата	Літера / Маса / Масштаб
Розроб.		Здота В.О.			
Перевір.		Калюжний О. О.			Аркуш 1 / Аркушів 1
Т. контр.					
Н. контр.		Сімоненко В.П.			НТУУ “КПІ” ФІОТ Група ІО-63
Затв.					



					ІАЛЦ.006310.006 Д3	
Зм.	Арк.	№ докум.	Підпис	Дата	Взаємодія частин системи Схема структурна	Літера / Маса / Масштаб
Розроб.		Здота В.О.				
Перевір.		Калюжний О.О.				Аркуш 1 / Аркушів 1
Т. контр.						
Н. контр.		Сімоненко В.П.				НТУУ "КПІ" ФІОТ Група ІО-63
Затв.						

ДОДАТОК Б

docker-compose.yml файл

```
version: '2.1'

services:

 rabbitmq:
  image: health_rabbitmq
  container_name:  moviefinder_rabbitmq
  build:
   context: .
   dockerfile: DockerRabbitMQ/Dockerfile
  ports:
   - '5674:5672'
   - '5675:5673'
   - '15672:15672'
  healthcheck:
   test: ["CMD",  "curl",  "-f", "http://localhost:15672"]
   interval: 20s
   timeout: 10s
   retries: 5
  hostname: rabbitmq

 moviefinder.webapi:
  image: ${DOCKER_REGISTRY}moviefinderwebapi
  container_name:  moviefinder_webapi
  build:
   context: .
   dockerfile: MovieFinder.WebAPI/Dockerfile
  ports:
   - "5005:80"
   - "443:443"
  depends_on:
   rabbitmq:
    condition:  service_healthy
  links:
```

```
    - rabbitmq
  volumes:
    - images:/opt/images
  hostname: moviefinder.webapi

  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - SECRET_KEY=${SECRET_KEY}
    - MOVIE_DB_CONNECTION=${MOVIE_DB_CONNECTION}
    - SCHEDULER_DB_CONNECTION=${SCHEDULER_DB_CONNECTION}
    -
ALLOWED_HOSTS=http://localhost:4200;http://localhost:88;http://localhost:8080;http://192.168.99.10
0:88
    - FILE_STORAGE_TYPE=${FILE_STORAGE_TYPE}
    - RabbitMQ__hostName=${RabbitMQ__hostName}
    - RabbitMQ__virtualHost=${RabbitMQ__virtualHost}
    - RabbitMQ__userName=${RabbitMQ__userName}
    - RabbitMQ__password=${RabbitMQ__password}

 moviefinder.notifier:
  image: ${DOCKER_REGISTRY}moviefindernotifier
  container_name: moviefinder_notifier
  build:
    context: .
    dockerfile: Services/Notifier/MovieFinder.Notifier.Service/Dockerfile
  ports:
    - "5009:80"
    - "543:443"
  depends_on:
    rabbitmq:
      condition: service_healthy
  links:
    - rabbitmq
  # volumes:
    # - ${APPDATA}/ASP.NET/Https:/root/.aspnet/https:ro
```

```
    # - ${APPDATA}/Microsoft/UserSecrets:/root/.microsoft/usersecrets:ro
    hostname: moviefinder.notifier
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      -
ALLOWED_HOSTS=http://localhost:4200;http://localhost:88;https://moviefinder.azurewebsites.net;http:
//192.168.99.100:88
      - RabbitMQ__hostName=${RabbitMQ__hostName}
      - RabbitMQ__virtualHost=${RabbitMQ__virtualHost}
      - RabbitMQ__userName=${RabbitMQ__userName}
      - RabbitMQ__password=${RabbitMQ__password}

  moviefinder.dataloader:
    image: ${DOCKER_REGISTRY}moviefinderdataloader
    container_name: moviefinder_dataloader
    build:
      context: .
      dockerfile: Services/DataLoader/MovieFinder.DataLoader.Service/Dockerfile
    depends_on:
      rabbitmq:
        condition: service_healthy
    links:
      - rabbitmq
    volumes:
      - images:/opt/images
    environment:
      - MOVIE_DB_CONNECTION=${MOVIE_DB_CONNECTION}
      - FILE_STORAGE_TYPE=${FILE_STORAGE_TYPE}
      - RabbitMQ__hostName=${RabbitMQ__hostName}
      - RabbitMQ__virtualHost=${RabbitMQ__virtualHost}
      - RabbitMQ__userName=${RabbitMQ__userName}
      - RabbitMQ__password=${RabbitMQ__password}

  moviefinder.scheduler:
    image: ${DOCKER_REGISTRY}moviefinderscheduler
```

```
  container_name:  moviefinder_scheduler
  build:
   context: .
   dockerfile: Services/Scheduler/MovieFinder.Scheduler.Service/Dockerfile
  depends_on:
   rabbitmq:
    condition:  service_healthy
  links:
   - rabbitmq
  environment:
   - SCHEDULER_DB_CONNECTION=${SCHEDULER_DB_CONNECTION}
   - RabbitMQ__hostName=${RabbitMQ__hostName}
   - RabbitMQ__virtualHost=${RabbitMQ__virtualHost}
   - RabbitMQ__userName=${RabbitMQ__userName}
   - RabbitMQ__password=${RabbitMQ__password}

 moviefinder.mailsender:
  image: ${DOCKER_REGISTRY}moviefindermailsender
  container_name:  moviefinder_mailsender
  build:
   context: .
   dockerfile: Services/MailSender/MovieFinder.MailSender.Service/Dockerfile
  depends_on:
   rabbitmq:
    condition:  service_healthy
  links:
   - rabbitmq
  environment:
   - MOVIE_DB_CONNECTION=${MOVIE_DB_CONNECTION}
   - SMTP_USERNAME=${SMTP_USERNAME}
   - SMTP_PASSWORD=${SMTP_PASSWORD}
   - RabbitMQ__hostName=${RabbitMQ__hostName}
   - RabbitMQ__virtualHost=${RabbitMQ__virtualHost}
   - RabbitMQ__userName=${RabbitMQ__userName}
   - RabbitMQ__password=${RabbitMQ__password}
```

```
moviefinder.usersubscriptions:
  image: ${DOCKER_REGISTRY}moviefinderusersubscriptions
  container_name:  moviefinder_usersubscriptions
  build:
    context: .
    dockerfile: Services/UserSubscription/MovieFinder.UserSubscriptions.Service/Dockerfile
  depends_on:
    rabbitmq:
      condition: service_healthy
  links:
    - rabbitmq
  environment:
    - MOVIE_DB_CONNECTION=${MOVIE_DB_CONNECTION}
    - MOVIES_LINK=${MOVIES_LINK}
    - RabbitMQ__hostName=${RabbitMQ__hostName}
    - RabbitMQ__virtualHost=${RabbitMQ__virtualHost}
    - RabbitMQ__userName=${RabbitMQ__userName}
    - RabbitMQ__password=${RabbitMQ__password}

moviefinder.client:
  image: ${DOCKER_REGISTRY}moviefinderclient
  container_name:  moviefinder_client
  build:
    context: ./MovieFinder.Client
    dockerfile: Dockerfile
  hostname: moviefinder.client
  ports:
    - 88:80
  depends_on:
    moviefinder.webapi:
      condition: service_started
  links:
    - moviefinder.webapi
  environment:
```

```
    - API_URL=http://192.168.99.100:5005/api
    - IMAGE_URL=http://192.168.99.100:5005/api/images
    - HUB_URL=http://192.168.99.100:5009/moviesHub
    - MOVIES_MESSAGE_METHOD=SendMovies

volumes:
 images:
  name: moviefinder_images
```

.NET код

```
using System;

namespace MovieFinder.Bus.Messages
{
   public abstract class BaseMessage
   {
      public DateTime CreatedDateTime { get; set; }

      public string CreatedBy { get; set; }
   }
}

using System;

namespace MovieFinder.Bus.Messages
{
   [Serializable]
   public class JobChangeMessage : BaseMessage
   {
      public bool IsUpdated { get; set; }
   }
}

using System;
using System.Collections.Generic;

namespace MovieFinder.Bus.Messages
{
   [Serializable]
   public class NewMoviesMessage : BaseMessage
   {

      public IEnumerable<int> MovieIds { get; set; }
   }
}

using System;
```

```
namespace MovieFinder.Bus.Messages
{
   [Serializable]
   public class SendMailsMessage  : BaseMessage
   {
      public bool ShouldSend  { get; set; }
   }
}

using System;

namespace MovieFinder.Bus.Messages
{

   [Serializable]
   public class StartLoadMovies : BaseMessage
   {
      public bool IsStart { get; set; }
   }
}

using System;
using System.Threading.Tasks;

namespace MovieFinder.Bus.Messengers.Interfaces
{
   /// <summary>
   /// Interface for subscribing  on messages
   /// </summary>
   public interface IMessageReceiver
   {
      /// <summary>
      /// Subscribes  on messages
      /// </summary>
      /// <typeparam name="T">Type  of object to which convert message</typeparam>
      /// <param name="exchangeName">Name  of exchange</param>
      /// <param name="queueName">Name  of queue</param>
      /// <param name="action">Callback  function on receiving  message</param>
      void SubscribeOnMessages<T>(string  exchangeName, string  queueName,  Func<T, Task> action)
where T : class;

   }
}

namespace MovieFinder.Bus.Messengers.Interfaces
{
   /// <summary>
   /// Interface for sending  messages
   /// </summary>
   public interface IMessageSender
   {
```

```
        /// <summary>
        /// Sends message
        /// </summary>
        /// <typeparam name="T">Type of message</typeparam>
        /// <param name="exchangeName">Name of exchange</param>
        /// <param name="routingKey">Name of queue</param>
        /// <param name="message">Message</param>
        void SendMessage<T>(string exchangeName, string routingKey, T message) where T : class;
    }
}

using System;
using System.Threading.Tasks;
using ActiveMqBroker.Infrastructure.Channel;
using ActiveMqBroker.Infrastructure.Exceptions;
using ActiveMqBroker.Infrastructure.Exchange;
using ActiveMqBroker.Infrastructure.Queue;
using ActiveMqBroker.Infrastructure.Serializers;
using ActiveMqBroker.Infrastructure.Settings;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using MovieFinder.Bus.Messengers.Interfaces;
using RabbitMQ.Client.Events;
using RabbitMQ.Client.Exceptions;

namespace ActiveMqBroker.Infrastructure.Messengers
{
    internal class AsyncMessageReceiver : IMessageReceiver
    {
        private readonly ILogger _logger;
        private readonly IProxyChannel _chanel;
        private readonly IQueueDeclarator _queueDeclarator;
        private RabbitMqSettings _settings;
        private readonly ISerializer _serializer;
        private readonly IExchangeDeclarator _exchangeDeclarator;

        public AsyncMessageReceiver(
            IOptions<RabbitMqSettings> options,
            IProxyChannel chanel,
            ILogger<AsyncMessageReceiver> logger,
            IQueueDeclarator queueDeclarator,
            IExchangeDeclarator exchangeDeclarator,
            ISerializer serializer)
        {
            _logger = logger ?? throw new ArgumentNullException(nameof(logger));
            _chanel = chanel ?? throw new ArgumentNullException(nameof(chanel));
            _settings = options?.Value ?? throw new ArgumentNullException(nameof(_settings));
            _serializer = serializer ?? throw new ArgumentNullException(nameof(serializer));
            _queueDeclarator = queueDeclarator ?? throw new
ArgumentNullException(nameof(queueDeclarator));
            _exchangeDeclarator = exchangeDeclarator ?? throw new
ArgumentNullException(nameof(exchangeDeclarator));
```

```
    }

    public void SubscribeOnMessages<T>(string exchangeName, string queueName, Func<T, Task>
action) where T : class
    {
      _logger.LogInformation("Start subscribing on messages ...");
      if (exchangeName == null) throw new ArgumentNullException(nameof(exchangeName));
      if (queueName == null) throw new ArgumentNullException(nameof(queueName));
      if (action == null) throw new ArgumentNullException(nameof(action));

      AsyncEventingBasicConsumer consumer;

      try
      {
        _exchangeDeclarator.DeclareExchange(exchangeName);
        _queueDeclarator.DeclareQueue(queueName);

        consumer = new AsyncEventingBasicConsumer(_chanel.GetInstance());
        _chanel.BasicConsume(queue: queueName, autoAck: false, consumer: consumer);
      }
      catch (RabbitMQClientException e)
      {
        _logger.LogError(e, "Error during creating consuming to queue {}", queueName);
        throw;
      }

      catch (BrokerSettingsException e)
      {
        {
          _logger.LogError(e, "Error during creating consuming to queue {}. Invalid settings",
queueName);
          throw;
        }
      }

      try
      {
        consumer.Received += async (o, ea) =>
        {

          T res = _serializer.Deserialize<T>(ea.Body);
          if (res.Equals(default(T)))
          {
            _logger.LogWarning("Json not valid for type {}", typeof(T));
            BasicAck(ea.DeliveryTag);
          }
          _logger.LogInformation("Json deserialized successfully...");
          await action.Invoke(res);
          _logger.LogDebug("Action performed");
          BasicAck(ea.DeliveryTag);
        };
      }
```

```
        catch (RabbitMQClientException e)
        {
          _logger.LogError(e, "Error during receiving message of type {} from queue {}", typeof(T),
queueName);
          throw;
        }
      }

      private void BasicAck(ulong deliveryTag)
      {
        _chanel.BasicAck(deliveryTag, false);
      }
    }
}

using System;
using System.Runtime.Serialization;
using ActiveMqBroker.Infrastructure.Channel;
using ActiveMqBroker.Infrastructure.Exchange;
using ActiveMqBroker.Infrastructure.Serializers;
using Microsoft.Extensions.Logging;
using MovieFinder.Bus.Messengers.Interfaces;

namespace ActiveMqBroker.Infrastructure.Messengers
{
  internal class MessageSender : IMessageSender
  {
    private readonly ILogger _logger;
    private readonly IProxyChannel _chanel;
    private readonly ISerializer _serializer;
    private readonly IExchangeDeclarator _exchangeDeclarator;

    public MessageSender(
      IProxyChannel chanel,
      ILogger<MessageSender> logger,
      ISerializer serializer,
      IExchangeDeclarator exchangeDeclarator)
    {
      _chanel = chanel ?? throw new ArgumentNullException(nameof(chanel));
      _logger = logger ?? throw new ArgumentNullException(nameof(logger));
      _serializer = serializer ?? throw new ArgumentNullException(nameof(serializer));
      _exchangeDeclarator = exchangeDeclarator ?? throw new
ArgumentNullException(nameof(exchangeDeclarator));
    }

    public void SendMessage<T>(string exchangeName, string routingKey, T message) where T : class
    {
      _logger.LogInformation("Start sending message {}...", typeof(T));

      if(exchangeName == null) throw new ArgumentNullException(nameof(exchangeName));
      if (routingKey == null) throw new ArgumentNullException(nameof(routingKey));
```

```
            _logger.LogDebug("Routing  key is {}, exchange is {}",routingKey,exchangeName);
            try
            {
               _exchangeDeclarator.DeclareExchange(exchangeName);
            }
            catch (Exception e)
            {
               _logger.LogError(e, "Error declaring exchange with name {}", exchangeName);
               throw;
            }

            byte[] messageBuffer;
            try
            {
               messageBuffer = _serializer.Serialize(message);
            }
            catch (SerializationException ex)
            {
               _logger.LogError(ex, "Error serializing object of type {}", typeof(T));
               throw;
            }

            try
            {
               _chanel.BasicPublish(exchangeName,
                  routingKey,
                  null,
                  messageBuffer);
            }
            catch (Exception ex)
            {
               _logger.LogError(ex, "Error during sending message of type {} to queue", typeof(T), routingKey);
               throw;
            }

            //TODO: serialize logs
            _logger.LogInformation("Sent {0} message", typeof(T));

         }
      }
   }

   namespace ActiveMqBroker.Infrastructure.Exchange
   {
      internal interface IExchangeDeclarator
      {
         void DeclareExchange(string name);
      }
   }

   using System;
   using System.Linq;
```

```
using ActiveMqBroker.Infrastructure.Channel;
using ActiveMqBroker.Infrastructure.Constants;
using ActiveMqBroker.Infrastructure.Exceptions;
using ActiveMqBroker.Infrastructure.Settings;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;

namespace ActiveMqBroker.Infrastructure.Exchange
{
   internal class ExchangeDeclarator : IExchangeDeclarator
   {
      private readonly ILogger _logger;
      private readonly IProxyChannel _chanel;
      private readonly RabbitMqSettings  _rabbitMqSettings;

      public ExchangeDeclarator(
         ILogger<ExchangeDeclarator> logger,
         IProxyChannel chanel,
         IOptions<RabbitMqSettings>  options)
      {
         _logger = logger ?? throw new ArgumentNullException(nameof(logger));
         _chanel = chanel ?? throw new ArgumentNullException(nameof(chanel));
         if (options == null) throw  new ArgumentNullException(nameof(options));
         _rabbitMqSettings  = options.Value;
      }

      public void DeclareExchange(string  name)
      {
         if(name == null) throw new ArgumentNullException(nameof(name));
         _logger.LogInformation("Start  declaring exchange with name {}...",name);

         if (name == ActiveMqConstants.DefaultExchangeName)
         {
            _logger.LogWarning("Exchange  type = \"\". Default exchange will be declared by broker
automatically");
            return;
         }

         ExchangeSettings  exchangeSettings  = _rabbitMqSettings.ExchangeSettings
            .FirstOrDefault(s  => s.Name ==  name);
         if (exchangeSettings  == null)
         {
            var msg = $"There is no exchange for name {name} in settings.";
            _logger.LogError(msg);
            throw new BrokerSettingsException(msg);
         }
         string type = exchangeSettings.Type.ToLower();

         if (type  != ActiveMqConstants.DirectExchange.ToLower()
            && type  != ActiveMqConstants.FanoutExchange.ToLower()
            && type  != ActiveMqConstants.TopicExchange.ToLower())
         {
```

```
            throw new BrokerSettingsException($"Exchange  type '{type}' is not known");
        }

        _chanel.ExchangeDeclare(name,
            type,
            exchangeSettings.Durable,
            exchangeSettings.AutoDelete,
            exchangeSettings.Arguments);
    }
  }
}

namespace ActiveMqBroker.Infrastructure.Queue
{
  /// <summary>
  /// Interface for declaring queues
  /// </summary>
  public interface IQueueDeclarator
  {
    /// <summary>
    /// Declares queue  for queueName
    /// </summary>
    /// <param name="queueName"></param>
    void DeclareQueue(string  queueName);
  }
}

using System;
using System.Linq;
using ActiveMqBroker.Infrastructure.Channel;
using ActiveMqBroker.Infrastructure.Exceptions;
using ActiveMqBroker.Infrastructure.Settings;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;

namespace ActiveMqBroker.Infrastructure.Queue
{
  internal class QueueDeclarator : IQueueDeclarator
  {
    private readonly ILogger _logger;
    private readonly RabbitMqSettings  _settings;
    private readonly IProxyChannel _chanel;

    public QueueDeclarator(
      IOptions<RabbitMqSettings>  options,
      ILogger<QueueDeclarator> logger,
      IProxyChannel chanel)
    {
      if (options == null) throw new ArgumentNullException(nameof(options));
      _chanel = chanel ?? throw new ArgumentNullException(nameof(chanel))  ;
      _logger = logger ?? throw new ArgumentNullException(nameof(logger));
      _settings = options.Value;
```

```
    }

    public void DeclareQueue(string queueName)
    {
      if (queueName == null) throw new ArgumentNullException(nameof(queueName));

      QueueSettings queueSettings;
      queueSettings = _settings.QueueSettings.FirstOrDefault(s => s.Name == queueName);
      if (queueSettings == null)
      {
        throw new BrokerSettingsException($"There is no settings for queue with name
{queueName}");
      }
      _logger.LogInformation("Declaring queue with name {} ...", queueSettings.Name);

      try
      {
        _chanel.QueueDeclare(
          queueSettings.Name,
          queueSettings.Durable,
          queueSettings.Exclusive,
          queueSettings.AutoDelete,
          queueSettings.Arguments);

        if (string.IsNullOrWhiteSpace(queueSettings.ExchangeName))
        {
          _logger.LogInformation("Binding to default exchange. Because {} = '{}'",
            nameof(queueSettings.ExchangeName), queueSettings.ExchangeName);
          return;
        }

        _chanel.QueueBind(
          queueName,
          queueSettings.ExchangeName,
          queueSettings.RoutingKey);
      }
      catch (Exception ex)
      {
        _logger.LogError(ex, "Error during creating queue with name {}", queueName);
        throw;
      }

    }
  }
}

namespace ActiveMqBroker.Infrastructure.Serializers
{
  /// <summary>
  /// Interface for serializing and deserializing object from string representation
  /// </summary>
  internal interface ISerializer
```

```
    {
        /// <summary>
        /// Deserializes object from byte array
        /// </summary>
        /// <typeparam name="T">Type of object to retrieve</typeparam>
        /// <param name="serializedObj">Byte representation of object</param>
        /// <returns>Desired object</returns>
        T Deserialize<T>(byte[] serializedObj) where T : class;

        /// <summary>
        /// Serializes object into byte array
        /// </summary>
        /// <typeparam name="T">Object type</typeparam>
        /// <param name="obj">Object ot serialize</param>
        /// <returns>Byte array representation of an object</returns>
        byte[] Serialize<T>(T obj) where T : class;
    }
}

using System.Text;
using ActiveMqBroker.Infrastructure.Exceptions;
using Newtonsoft.Json;

namespace ActiveMqBroker.Infrastructure.Serializers
{
    /// <summary>
    /// Json Serializer
    /// </summary>
    internal class JsonSerializer : ISerializer
    {
        public T Deserialize<T>(byte[] serializedObj) where T: class
        {
            var strObject = Encoding.UTF8.GetString(serializedObj);
            strObject = strObject.Trim();
            if (!strObject.IsValidJson())
                throw new BrokerMessageDeserializingException("Object can't be deserialized as JSON");
            T obj = JsonConvert.DeserializeObject<T>(strObject);
            return obj;

        }

        public byte[] Serialize<T>(T obj) where T : class
        {
            string jsonObject = JsonConvert.SerializeObject(obj);
            return Encoding.UTF8.GetBytes(jsonObject);
        }
    }
}

using ActiveMqBroker.Infrastructure.Channel;
using ActiveMqBroker.Infrastructure.Exceptions;
using ActiveMqBroker.Infrastructure.Exchange;
```

```
using ActiveMqBroker.Infrastructure.Messengers;
using ActiveMqBroker.Infrastructure.Queue;
using ActiveMqBroker.Infrastructure.Serializers;
using ActiveMqBroker.Infrastructure.Settings;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using MovieFinder.Bus.Messengers.Interfaces;
using RabbitMQ.Client;

namespace ActiveMqBroker.Infrastructure
{
   /// <summary>
   /// Extension methods to add RabbitMq to IoC
   /// </summary>
   public static class RabbitMqIocExtensions
   {
      private const string DefaultUsername = "guest";
      private const string DefaultPassword = "guest";
      private const string DefaultHostName = "localhost";
      private const string DefaultVirtualHost = "/";

      /// <summary>
      /// Register RabbitMq using IHostBuilder
      /// </summary>
      /// <param name="builder">Host builder</param>
      /// <param name="rabbitMqSettingsName">Name of settings section for RabbitMQ</param>
      /// <returns>Builder</returns>
      public static IHostBuilder AddDefaultRabbitMq(this IHostBuilder builder, string
rabbitMqSettingsName)
      {
         return builder.ConfigureServices((hostContext, services) =>
         {
            IConfiguration configuration = hostContext.Configuration;

            services.AddDefaultRabbitMq(configuration, rabbitMqSettingsName);
         });
      }

      /// <summary>
      /// Register RabbitMQ in IServices collection
      /// </summary>
      /// <param name="services">Collection of dependencies</param>
      /// <param name="configuration">Configuration of application or service</param>
      /// <param name="rabbitMqSettingsName">Name of settings section for RabbitMQ</param>
      public static void AddDefaultRabbitMq(this IServiceCollection services, IConfiguration configuration,
string rabbitMqSettingsName)
      {

         IConfigurationSection rabbitSections = configuration.GetSection(rabbitMqSettingsName);

         if (!rabbitSections.Exists())
```

```
        {
            throw new BrokerSettingsException($"{nameof(rabbitMqSettingsName)} Must be present in
settings");
        }

        services.Configure<RabbitMqSettings>(s => rabbitSections.Bind(s));

        RabbitMqSettings rabbitSettings = rabbitSections.Get<RabbitMqSettings>();

        services.AddSingleton<IConnection>(provider =>
        {
            ConnectionFactory connectionFactory = new ConnectionFactory
            {
                VirtualHost = rabbitSettings.VirtualHost ?? DefaultVirtualHost,
                HostName = rabbitSettings.HostName ?? DefaultHostName,
                UserName = rabbitSettings.UserName ?? DefaultUsername,
                Password = rabbitSettings.Password ?? DefaultPassword,
                DispatchConsumersAsync = rabbitSettings.IsAsyncConsume
            };
            return connectionFactory.CreateConnection();
        });

        services.AddSingleton<IQueueDeclarator, QueueDeclarator>();

        services.AddSingleton<IModel>(provider
            => provider
                .GetRequiredService<IConnection>()
                .CreateModel());

        services.AddSingleton<IProxyChannel, ProxyRabbitChannel>();

        if (rabbitSettings.IsAsyncConsume)
        {
            services.AddSingleton<IMessageReceiver, AsyncMessageReceiver>();
        }
        else
        {
            services.AddSingleton<IMessageReceiver, MessageReceiver>();
        }

        services.AddScoped<IMessageSender, MessageSender>();

        services.AddSingleton<ISerializer, JsonSerializer>();

        services.AddSingleton<IExchangeDeclarator, ExchangeDeclarator>();
    }


  }
}

using System;
```

```
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using MovieFinder.Bus.Messages;
using MovieFinder.Bus.Messengers.Interfaces;
using MovieFinder.Scheduler.Settings;
using Quartz;

namespace MovieFinder.Scheduler.Jobs
{
   /// <summary>
   /// Job for notification to load data
   /// </summary>
   internal class DataLoaderNotificatorJob : IJob
   {
      private readonly ILogger _logger;
      private readonly IMessageSender _messageSender;
      private readonly BrokerSettings _brokerSettings;
      public DataLoaderNotificatorJob(
         IMessageSender messageSender,
         ILogger<DataLoaderNotificatorJob> logger,
         IOptions<BrokerSettings>  queueOptions)
      {
         _logger = logger ?? throw new ArgumentNullException(nameof(logger));
         _messageSender  = messageSender  ?? throw new
ArgumentNullException(nameof(messageSender));
         _brokerSettings  = queueOptions.Value  ?? throw new
ArgumentNullException(nameof(_brokerSettings));
      }

      public Task Execute(IJobExecutionContext context)
      {
         _logger.LogInformation("{} job executing...", nameof(DataLoaderNotificatorJob));
         _messageSender.SendMessage(
            _brokerSettings.DataLoadingExchangeName,
            _brokerSettings.DataLoadingRoutingKey,  new StartLoadMovies { IsStart = true });
         _logger.LogInformation("Message  is sent");
         return Task.CompletedTask;
      }
   }
}

using System.Threading.Tasks;
using MovieFinder.Bus.Messages;

namespace MovieFinder.Scheduler.MessageHandlers
{
   /// <summary>
   /// Interface for message handling
   /// </summary>
   internal interface IReceivedMessageHandler
   {
```

```
        /// <summary>
        /// Method to perform on receiving message
        /// </summary>
        /// <param name="jobChangeMessage">Message</param>
        /// <returns></returns>
        Task OnJobChangeReceived(JobChangeMessage jobChangeMessage);
    }
}

using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;
using MovieFinder.Bus.Messages;
using MovieFinder.Scheduler.Workers;

namespace MovieFinder.Scheduler.MessageHandlers
{
    internal class ReceivedMessageHandler : IReceivedMessageHandler
    {
        private readonly ILogger _logger;
        private readonly IJobRunner _jobRunner;

        public ReceivedMessageHandler(
            ILogger<ReceivedMessageHandler> logger,
            IJobRunner jobRunner)
        {
            _logger = logger ?? throw new ArgumentNullException(nameof(logger));
            _jobRunner = jobRunner ?? throw new ArgumentNullException(nameof(jobRunner));
        }

        public async Task OnJobChangeReceived(JobChangeMessage jobChangeMessage)
        {
            _logger.LogInformation("Message of type {} received with value for CreatedDateTime = {}",
                typeof(JobChangeMessage),
                jobChangeMessage.CreatedDateTime);
//          if (jobChangeMessage.IsUpdated)
//          {
                DateTime now = DateTime.Now;
                TimeSpan ts = now - jobChangeMessage.CreatedDateTime;
//              if(ts.TotalHours > 1)
                _logger.LogInformation("Executing refreshing jobs...");
                await _jobRunner.RefreshJobs();
//          }
        }
    }
}

using System;

namespace MovieFinder.Scheduler.Models
{
    /// <summary>
```

```
    /// Model for quartz job
    /// </summary>
    internal class JobSchedulerModel
    {
        /// <summary>
        /// Id of job
        /// </summary>
        public int Id { get; set; }

        /// <summary>
        /// Name of job
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// Description of job
        /// </summary>
        public string Description { get; set; }

        /// <summary>
        /// Trigger for job
        /// </summary>
        public TriggerSchedulerModel  QuartzTrigger { get; set; }

        /// <summary>
        /// .Net Type of job
        /// </summary>
        public Type JobType { get; set; }
    }
}

namespace MovieFinder.Scheduler.Models
{
    /// <summary>
    /// Model for quartz Trigger
    /// </summary>
    internal class TriggerSchedulerModel
    {
        /// <summary>
        /// Id of trigger
        /// </summary>
        public int Id { get; set; }

        /// <summary>
        /// Name of trigger
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// Description of trigger
        /// </summary>
        public string Description { get; set; }
```

```
        /// <summary>
        /// Cron for scheduling job
        /// </summary>
        public string Cron { get; set; }
    }
}

using System.Collections.Generic;
using System.Threading.Tasks;
using MovieFinder.Scheduler.Models;

namespace MovieFinder.Scheduler.Workers
{
    /// <summary>
    /// Interface for retrieving jobs from storage
    /// </summary>
    internal interface IJobLoader
    {
        /// <summary>
        /// Loads jobs from storage
        /// </summary>
        /// <returns></returns>
        Task<IEnumerable<JobSchedulerModel>> LoadJobs();
    }
}

using System.Threading.Tasks;

namespace MovieFinder.Scheduler.Workers
{
    /// <summary>
    /// Interface for running  jobs
    /// </summary>
    internal interface IJobRunner
    {
        /// <summary>
        /// Runs jobs
        /// </summary>
        /// <returns></returns>
        Task RunJobs();

        /// <summary>
        /// Refreshes jobs scheduling
        /// </summary>
        /// <returns></returns>
        Task RefreshJobs();
    }
}

using System.Threading.Tasks;
using MovieFinder.Scheduler.Models;
```

```
using Quartz;

namespace MovieFinder.Scheduler.Workers
{
  internal interface ITriggerRenewer
  {
    Task<ITrigger> RenewTrigger(string oldTriggerName, TriggerSchedulerModel newTrigger);
  }
}

using System;

namespace MovieFinder.Scheduler.Workers
{
  internal interface ITypeConvertor
  {
    Type ConvertToType(string name);
  }
}

using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;
using MovieFinder.Scheduler.Jobs;
using MovieFinder.Scheduler.Jobs.Models;
using MovieFinder.Scheduler.Models;

namespace MovieFinder.Scheduler.Workers
{
  internal class JobLoader : IJobLoader
  {
    private readonly ILogger _logger;
    private readonly IJobService _jobService;
    private readonly ITypeConvertor _typeConvertor;

    public JobLoader(
      IJobService jobService,
      ITypeConvertor typeConvertor,
      ILogger<JobLoader> logger)
    {
      _jobService = jobService ?? throw new ArgumentNullException(nameof(jobService));
      _typeConvertor = typeConvertor ?? throw new ArgumentNullException(nameof(jobService));
      _logger = logger ?? throw new ArgumentNullException(nameof(logger));

    }

    public async Task<IEnumerable<JobSchedulerModel>> LoadJobs()
    {
      _logger.LogInformation("Start loading jobs...");
      try
      {
```

```
            _logger.LogInformation("Retrieving all jobs ...");
            IEnumerable<JobModel> jobs = await _jobService.GetAll();
            ISet<JobSchedulerModel> jobsToExecute = new HashSet<JobSchedulerModel>();
            foreach (JobModel job in jobs)
            {
                _logger.LogInformation("Processing job {}", job.Name);
                JobSchedulerModel jobModel = new JobSchedulerModel();
                try
                {

                    Type jobType = _typeConvertor.ConvertToType(job.Name);

                    if (job.QuartzTrigger == null)
                    {
                        throw new ArgumentNullException(nameof(job.QuartzTrigger));
                    }
                    jobModel.Id = job.Id;
                    jobModel.JobType = jobType;
                    jobModel.Name = job.Name;
                    jobModel.Description = job.Description;

                    jobModel.QuartzTrigger = new TriggerSchedulerModel
                    {
                        Id = job.QuartzTrigger.Id,
                        Name = job.QuartzTrigger.Name,
                        Description = job.QuartzTrigger.Description,
                        Cron = job.QuartzTrigger.Cron
                    };

                    jobsToExecute.Add(jobModel);
                    _logger.LogInformation("Job added successfully {}", jobModel.Name);
                }
                catch (TypeLoadException e)
                {
                    _logger.LogError(e, "Error loading type {}", jobModel.Name);
                }
            }

            _logger.LogInformation("Finishing processing jobs...");
            return jobsToExecute;
        }
        catch(Exception ex)
        {
            _logger.LogError(ex, "Exception during loading jobs");
            throw;
        }
    }
  }
}

using System;
```

```
namespace MovieFinder.Scheduler.Workers
{
   internal class JobTypeConvertor : ITypeConvertor
   {
      private const string JobsFolder = "MovieFinder.Scheduler.Jobs";

      public Type ConvertToType(string name)
      {
         Type jobType = Type.GetType($"{JobsFolder}.{name}", true);
         return jobType;
      }
   }
}

using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;
using MovieFinder.Scheduler.Exceptions;
using MovieFinder.Scheduler.Models;
using Quartz;

namespace MovieFinder.Scheduler.Workers
{
   internal class TriggerRenewer : ITriggerRenewer
   {
      private readonly IScheduler _scheduler;
      private readonly ILogger _logger;

      public TriggerRenewer(
         IScheduler scheduler,
         ILogger<TriggerRenewer> logger)
      {
         _scheduler = scheduler;
         _logger = logger;
      }
      public async Task<ITrigger> RenewTrigger(string oldTriggerName, TriggerSchedulerModel
newTrigger)
      {
         if (oldTriggerName == null) throw new ArgumentNullException(nameof(oldTriggerName));
         if (newTrigger == null) throw new ArgumentNullException(nameof(newTrigger));

         _logger.LogInformation("Start renewing trigger with name {} ... ", oldTriggerName);

         ITrigger oldTrigger = await _scheduler.GetTrigger(new TriggerKey(oldTriggerName));
         if(oldTrigger == null)
         {
            throw new TriggerNotFoundException($"Trigger with name {oldTriggerName} not found");
         }

         TriggerBuilder tb = oldTrigger.GetTriggerBuilder();

         ITrigger newTriggerSched = tb
```

```
            .WithIdentity(newTrigger.Name)
            .WithDescription(newTrigger.Description)
            .WithCronSchedule(newTrigger.Cron).Build();

        return newTriggerSched;
      }
    }
}

using System;
using Microsoft.Extensions.DependencyInjection;
using Quartz;
using Quartz.Spi;

namespace MovieFinder.Scheduler
{
    /// <summary>
    /// JobFactory for working with singletons from serviceProvider
    /// </summary>
    internal class SingletonJobFactory : IJobFactory
    {
        private readonly IServiceProvider _serviceProvider;

        public SingletonJobFactory(IServiceProvider serviceProvider)
        {
            _serviceProvider = serviceProvider ?? throw new
ArgumentNullException(nameof(serviceProvider));
        }

        public IJob NewJob(TriggerFiredBundle bundle, IScheduler scheduler)
        {
            return _serviceProvider.GetRequiredService(bundle.JobDetail.JobType) as IJob;
        }

        public void ReturnJob(IJob job)
        {
        }
    }
}

using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using MovieFinder.Bus.Messages;
using MovieFinder.Bus.Messengers.Interfaces;
using MovieFinder.Scheduler.MessageHandlers;
using MovieFinder.Scheduler.Settings;
using MovieFinder.Scheduler.Workers;
using Quartz;
```

```
namespace MovieFinder.Scheduler
{
   internal class Starter
   {
      private readonly ILogger _log;
      private readonly IJobRunner _jobRunner;
      private readonly IScheduler _scheduler;
      private readonly IMessageReceiver _messageReceiver;
      private readonly IReceivedMessageHandler _receivedMessageHandler;
      private readonly BrokerSettings _queueBindSettings;

      public Starter(
         ILogger<Starter> log,
         IJobRunner jobRunner,
         IScheduler scheduler,
         IMessageReceiver messageReceiver,
         IReceivedMessageHandler receivedMessageHandler,
         IOptions<BrokerSettings> bindOptions)
      {
         _log = log ?? throw new ArgumentNullException(nameof(log));
         _scheduler = scheduler ?? throw new ArgumentNullException(nameof(scheduler));
         _messageReceiver = messageReceiver ?? throw new
ArgumentNullException(nameof(messageReceiver));
         _receivedMessageHandler = receivedMessageHandler ?? throw new
ArgumentNullException(nameof(receivedMessageHandler));
         _jobRunner = jobRunner ?? throw new ArgumentNullException(nameof(jobRunner));
         _queueBindSettings = bindOptions.Value ?? throw new
ArgumentNullException(nameof(_queueBindSettings));
      }

      public async Task Start(CancellationToken cancellationToken)
      {
         try
         {
            _log.LogInformation("Starting...");

            await _jobRunner.RunJobs();

            _log.LogInformation("Subscribing on messages...");

_messageReceiver.SubscribeOnMessages<JobChangeMessage>(_queueBindSettings.JobsExchangeNam
e,
               _queueBindSettings.JobQueueName,
               async m => await _receivedMessageHandler.OnJobChangeReceived(m)
            );

            _log.LogInformation("Starting scheduler ...");
            await _scheduler.Start(cancellationToken);

            _log.LogInformation("Start preparations are done. Service started");
         }
         catch (Exception ex)
```

```
        {
            _log.LogError(ex, "Error during starting");
            throw;
        }
    }

    public async Task Shutdown(CancellationToken  cancellationToken)
    {
        await  _scheduler.Shutdown(cancellationToken);
    }
  }
}

using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace MovieFinder.Scheduler.Service
{
  /// <summary>
  /// Hosted service for managing quartz jobs
  /// </summary>
  internal class QuartzSchedulerService  : IHostedService
  {
    private readonly ILogger<QuartzSchedulerService>  _log;
    private readonly Starter _starter;

    public QuartzSchedulerService(
      ILogger<QuartzSchedulerService>  log,
      Starter starter)
    {
      _log = log ?? throw new ArgumentNullException(nameof(log));
      _starter = starter ?? throw  new ArgumentNullException(nameof(starter));
    }


    public async Task StartAsync(CancellationToken  cancellationToken)
    {
      try
      {
        _log.LogInformation("QuartzSchedulerService  starting...");

        await _starter.Start(cancellationToken);
      }catch(Exception ex)
      {
        _log.LogError(ex, "Error during starting service");
        throw;
      }
    }
```

```
        public async Task StopAsync(CancellationToken  cancellationToken)
        {
            _log.LogInformation("{} stopping...",nameof(QuartzSchedulerService));

            await _starter.Shutdown(cancellationToken);

            _log.LogInformation("{} stopped",  nameof(QuartzSchedulerService));
        }
    }
}

using System.Threading.Tasks;
using ActiveMqBroker.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using MovieFinder.CoreServiceBase.Extensions;
using MovieFinder.Scheduler.DAL;
using MovieFinder.Scheduler.DAL.Repositories;
using MovieFinder.Scheduler.Jobs;
using MovieFinder.Scheduler.MessageHandlers;
using MovieFinder.Scheduler.Settings;
using MovieFinder.Scheduler.Workers;
using Quartz;
using Quartz.Impl;
using Quartz.Spi;

namespace MovieFinder.Scheduler.Service
{
    internal class Program
    {
        private const string SchedulerDatabaseConnectionString = "SchedulerDbConnection";
        private const string RabbitMqSettings = "RabbitMQ";
        private const string BrokerSettings = "BrokerSettings";
        private const string SchedulerDbConnectionVar = "SCHEDULER_DB_CONNECTION";

        static async Task Main(string[] args)
        {
            IHostBuilder builder = GetBuilder();

            await builder.RunConsoleAsync();

        }

        private static IHostBuilder GetBuilder()
        {
            return new HostBuilder()
                .CreateDefaultHostBuilder()
                .AddDefaultRabbitMq(RabbitMqSettings)
                .ConfigureServices((hostContext,  services) =>
```

```
        {
            string connectionString = hostContext.Configuration[SchedulerDbConnectionVar]
                                ?? hostContext.Configuration.GetConnectionString(
                                  SchedulerDatabaseConnectionString);


            services.AddDbContext<SchedulerDbContext>(options => options.UseSqlServer(
              connectionString,
              o => o.EnableRetryOnFailure())
              .ConfigureWarnings(warnings => warnings.Throw(
                 RelationalEventId.QueryClientEvaluationWarning)));

            services.AddLogging();
            services.AddHostedService<QuartzSchedulerService>();

            services.AddScoped<IJobRepository, JobRepository>();

            services.AddScoped<IJobFactory, SingletonJobFactory>();

            services.AddSingleton<ISchedulerFactory, StdSchedulerFactory>();

            services.AddSingleton<IJobService, JobService>();

            services.AddScoped<IJobLoader, JobLoader>();

            services.AddSingleton<IJobRunner, JobRunner>();

            services.AddSingleton<ITypeConvertor, JobTypeConvertor>();

            services.AddSingleton<ITriggerRenewer, TriggerRenewer>();

            services.AddSingleton<IScheduler>(provider =>
            {
              IScheduler scheduler = provider.GetRequiredService<ISchedulerFactory>()
                 .GetScheduler()
                 .Result;
              scheduler.JobFactory = provider.GetRequiredService<IJobFactory>();
              return scheduler;
            });

            IConfigurationSection brokerSettingsSection =
hostContext.Configuration.GetSection(BrokerSettings);
            services.Configure<BrokerSettings>(s => brokerSettingsSection.Bind(s));

            services.AddSingleton<DataLoaderNotificatorJob>();

            services.AddSingleton<IReceivedMessageHandler, ReceivedMessageHandler>();

            services.AddSingleton<Starter>();

        });
    }
```

```
    }
}

using System.Threading.Tasks;
using MovieFinder.Bus.Messages;

namespace MovieFinder.MailSender.MessageHandlers
{
    /// <summary>
    /// Interface of messsage handler
    /// </summary>
    internal interface IReceivedMessageHandler
    {
        /// <summary>
        /// Method for handling incoming messages
        /// </summary>
        /// <param name="sendMailsMessage">Message model</param>
        /// <returns></returns>
        Task OnEmailsSendReceived(SendMailsMessage sendMailsMessage);
    }
}

using System;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using MovieFinder.Bus.Messages;
using MovieFinder.MailSender.Services;
using MovieFinder.MailSender.Settings;

namespace MovieFinder.MailSender.MessageHandlers
{
    internal class ReceivedMessageHandler : IReceivedMessageHandler
    {
        private readonly ILogger _logger;
        private BrokerSettings _queuesBindSettings;
        //private readonly IMailProcessor _mailProcessor;
        private readonly IServiceProvider _serviceProvider;

        public ReceivedMessageHandler(
            ILogger<ReceivedMessageHandler> logger,
            //IMailProcessor mailProcessor,
            IOptions<BrokerSettings> queuesBindOptions, IServiceProvider serviceProvider)
        {
            _logger = logger ?? throw new ArgumentNullException(nameof(logger));
            //_mailProcessor = mailProcessor ?? throw new ArgumentNullException(nameof(logger));
            _serviceProvider = serviceProvider ?? throw new
ArgumentNullException(nameof(serviceProvider));
            _queuesBindSettings = queuesBindOptions?.Value ?? throw new
ArgumentNullException(nameof(_queuesBindSettings));
        }
```

```
        public async Task OnEmailsSendReceived(SendMailsMessage  sendMailsMessage)
        {
            try
            {
                _logger.LogInformation("Message  of type {} received. ShouldSent  = {}",
typeof(JobChangeMessage),  sendMailsMessage.ShouldSend);

                if (sendMailsMessage.ShouldSend)
                {
                    using (IServiceScope scope = _serviceProvider.CreateScope())
                    {
                        IMailProcessor mailProcessor = scope.ServiceProvider.GetService<IMailProcessor>();
                        await mailProcessor.Send();
                    }
                    //await _mailProcessor.Send();
                }
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Error during receiving message..");
                throw;
            }
        }
    }
}

using System;
using System.Linq;
using Castle.Core.Internal;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using MimeKit;
using MimeKit.Text;
using MovieFinder.MailSender.CustomSmtp;
using MovieFinder.MailSender.Models;
using MovieFinder.MailSender.Settings;
using Newtonsoft.Json;

namespace MovieFinder.MailSender.Services
{
    internal class EmailSender : IMailSender , IDisposable
    {
        private readonly EmailSettings _emailSettings;
        private readonly ILogger _logger;
        private readonly ISmtpClient _smtpClient;

        public EmailSender(IOptions<EmailSettings> options, ILogger<EmailSender> logger, ISmtpClient
smtpClient)
        {
            if(options == null) throw new ArgumentNullException(nameof(options));
            _logger = logger ?? throw new ArgumentNullException(nameof(logger));
```

```
        _smtpClient = smtpClient ?? throw new ArgumentNullException(nameof(smtpClient));

        _emailSettings = options.Value;
    }

    public void Send(EmailMessage emailMessage)
    {
        if(emailMessage == null) throw new ArgumentNullException(nameof(emailMessage));
        if (emailMessage.FromAddresses.IsNullOrEmpty())
            throw new ArgumentException($"{nameof(emailMessage.FromAddresses)} can't be empty");
        if (emailMessage.ToAddresses.IsNullOrEmpty())
            throw new ArgumentException($"{nameof(emailMessage.ToAddresses)} can't be empty");

        _logger.LogInformation("Sending email:{}...", JsonConvert.SerializeObject(emailMessage));

        MimeMessage message = new MimeMessage();

        message.To.AddRange(emailMessage.ToAddresses.Select(x => new MailboxAddress(x.Name,
x.Address)));
        message.From.AddRange(emailMessage.FromAddresses.Select(x => new
MailboxAddress(x.Name, x.Address)));

        message.Subject = emailMessage.Subject;

        message.Body = new TextPart(TextFormat.Html)
        {
            Text = emailMessage.Content
        };

        try
        {
            if (!_smtpClient.IsConnected())
            {
                _logger.LogDebug("Connecting to SMTP server");
                _smtpClient.Connect(_emailSettings.SmtpServer, _emailSettings.SmtpPort,
_emailSettings.UseSsl);
            }
        }
        catch (Exception e)
        {
            _logger.LogError(e,"Error during connecting to smtp client");
            throw;
        }

        try
        {
            if (!_smtpClient.IsAuthenticated())
            {
                _logger.LogDebug("Authenticating to SMTP server for user {}",
                    _emailSettings.SmtpUsername);
                _smtpClient.Authenticate(_emailSettings.SmtpUsername, _emailSettings.SmtpPassword);
```

```
            }
        }
        catch (Exception e)
        {
            _logger.LogError(e, "Error during authenticating to smtp client");
            throw;
        }

        try
        {
            _smtpClient.Send(message);

        }
        catch (Exception e)
        {
            _logger.LogError(e, "Error during sending email to
{}",JsonConvert.SerializeObject(emailMessage.ToAddresses));
            throw;
        }


    }

    public void Dispose()
    {
        _smtpClient.Disconnect(true);
    }
  }
}

using System;
using System.Collections.Generic;
using System.Data.Common;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using MovieFinder.Mail.Reader;
using MovieFinder.Mail.Writer;
using MovieFinder.MailSender.Models;
using MovieFinder.MailSender.Settings;
using MovieFinder.Models;

namespace MovieFinder.MailSender.Services
{
  internal class MailProcessor : IMailProcessor
  {
    private readonly IMailReaderService _mailReaderService;
    private readonly IMailWriterService _mailWriterService;
    private readonly ILogger _logger;
    private readonly IMailSender _mailSender;
    private readonly EmailSettings _settings;
```

```
        public MailProcessor(
            IMailReaderService mailReaderService,
            IMailWriterService mailWriterService,
            ILogger<MailProcessor> logger,
            IMailSender mailSender,
            IOptions<EmailSettings> options)
        {
            _mailReaderService = mailReaderService ?? throw new
ArgumentNullException(nameof(mailReaderService));
            _logger = logger ?? throw new ArgumentNullException(nameof(logger));
            _mailSender = mailSender ?? throw new ArgumentNullException(nameof(mailSender));
            _mailWriterService = mailWriterService ?? throw new
ArgumentNullException(nameof(mailWriterService));
            if (options == null) throw new ArgumentNullException(nameof(options));

            _settings = options.Value;
        }
        public async Task Send()
        {
            _logger.LogInformation("Start sending messages ...");
            IEnumerable<MailMessage> mailMessages;
            try
            {
                mailMessages = await _mailReaderService.GetAllUnsent();
            }
            catch (DbException e)
            {
                _logger.LogInformation(e, "Error during getting unsent mails");
                throw;
            }

            _logger.LogDebug("Number of unsent messages: {}", mailMessages.Count());

            foreach (var m in mailMessages)
            {
                EmailMessage message = new EmailMessage
                {
                    Content = m.Text,
                    Subject = m.Subject
                };

                message.FromAddresses.Add(new EmailAddress
                {
                    Address = _settings.Sender,
                    Name = _settings.SenderName
                });

                message.ToAddresses.Add(new EmailAddress
                {
                    Address = m.User.Email,
                    Name = m.User.Email
```

```
            });

            _logger.LogInformation("Sending  message {} to {} ...",m.Id, m.User.Email);
            try
            {
               _mailSender.Send(message);
            }
            catch (Exception e)
            {
               _logger.LogError(e, "Error during sending mail {} directed to {}", m.Id, m.User.Email);
               throw;
            }

            try
            {
               await _mailWriterService.SetSent(m.Id);
            }
            catch (Exception e)
            {
               _logger.LogError(e, "Error during setting status to sent of email {} directed to {}",
                  m.Id, m.User.Email);
               throw;
            }
            _logger.LogInformation("Message  {} was sent to {}", m.Id, m.User.Email);
         }
      }
   }
}
```